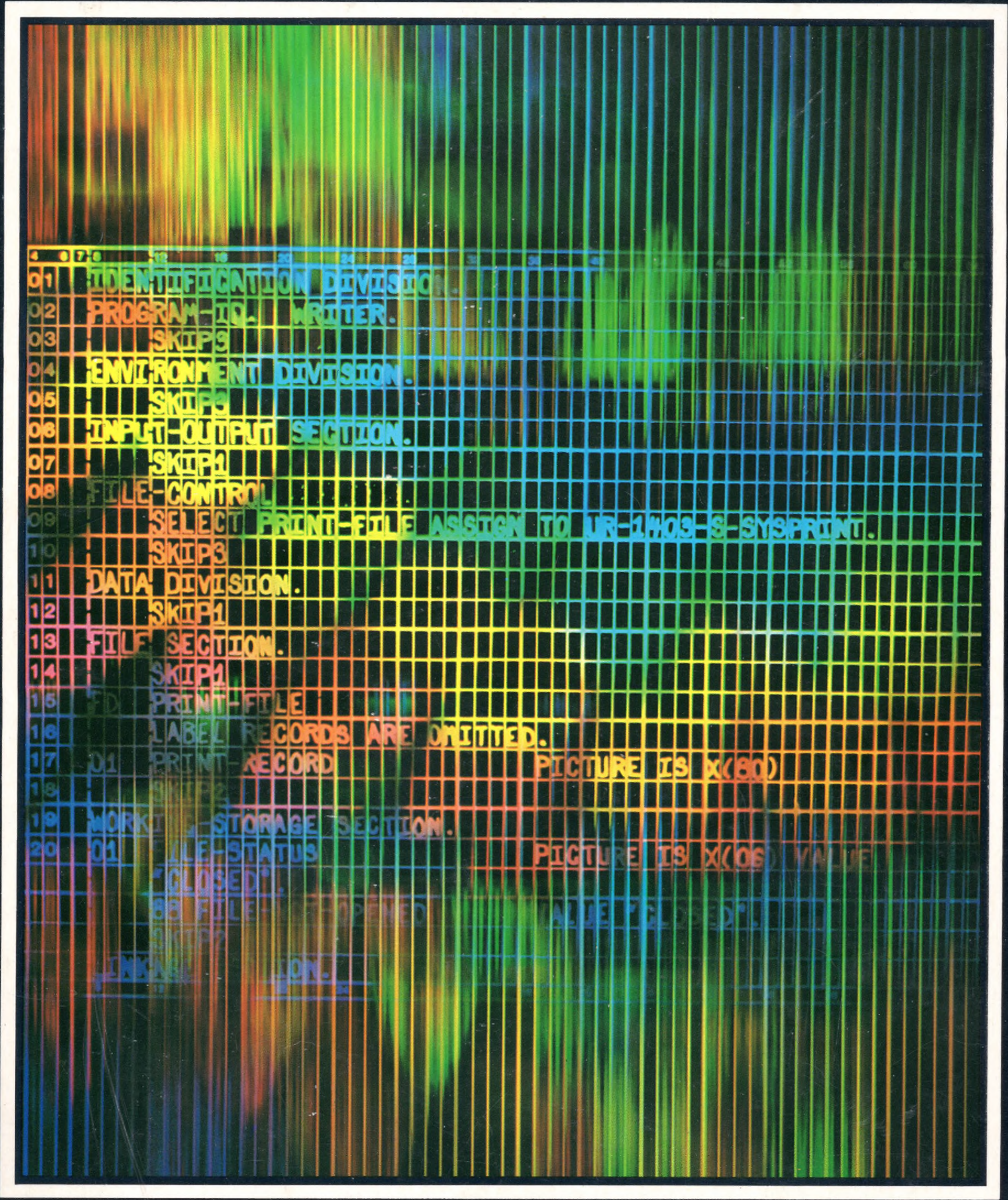
Introductory Structured COBOL

A Programming Skills Approach



Howard D. Weiner

wcb

Introductory Structured COBOL

Introductory Structured COBOL

A Programming Skills Approach

Howard D. Weiner
The American University,
Washington D.C.

wcb
Wm. C. Brown Company Publishers
Dubuque, Iowa

Book Team
Robert B. Stern   Senior Editor
Nicholas Murray   Associate Developmental Editor
Barbara J. Grantham   Designer
Patricia M. Schmelling   Production Editor

wcb group
Wm. C. Brown   Chairman of the Board
Mark C. Falb   Executive Vice President

wcb
Wm. C. Brown Company Publishers, College Division
Lawrence E. Cremer   President
David Wm. Smith   Vice President, Marketing
E. F. Jogerst   Vice President, Cost Analyst
David A. Corona   Assistant Vice President, Production Development and Design
James L. Romig   Executive Editor
Marcia H. Stout   Marketing Manager
Janis Machala   Director of Marketing Research
Marilyn A. Phelps   Manager of Design
William A. Moss   Production Editorial Manager

Cover photo by Bob Coyle

Library of Congress Catalog Card Number: 82-71567

ISBN 0-697-08149-4

2-08149-01

Printed in the United States of America

Contents

Preface

Purpose

This book arose from the conviction that it was possible to teach computer programming using the COBOL language in such a way that students could focus on developing programming skills without being overwhelmed and discouraged by the detail of the COBOL language itself. It aims to stimulate concern for the importance of sound program design and presents COBOL as the final, formal expression of carefully thought out problem-solving strategies. The gradual introduction of COBOL keeps students confident of their mastery of the language, and the emphasis on design develops essential programming skills from the outset.

Most COBOL texts, in less than their first one hundred pages, introduce virtually the entire subset of the COBOL language commonly used in programs manipulating sequential files. At precisely the time when students are grappling with the process of programming, they are also expected to master a significant portion of the COBOL language. This book avoids making such demands, and the result is an eminently teachable presentation of the design, implementation, and maintenance of business transaction processing programs from a unique perspective.

Prerequisites

The text is intended for a first course in COBOL programming. It assumes no prior knowledge of any other programming language. It does, however, assume that students are aware of the basic concepts of data processing as presented in standard introductory courses that survey the field. Another text is planned to include topics suited to a second, advanced course in COBOL programming.

Approach

In a brilliant essay, Charles Lindbloom noted that people are incremental problem solvers, not rational ones. We try to categorize problems by known types and then use known, proven solution strategies to solve them.

The beginning programmer faces the worst of all possible circumstances. The problems are completely new, and the methods of solving them are unlike (or so it appears) those used for more familiar problems. Beginners are searching for (1) a way of recognizing and classifying like problems and (2) a solution for each set of problems. They need to know that they do not have to perceive programming as fundamentally different from their other experiences.

Classification of Programs

Accordingly, the approach taken in this text is to classify programs into four different types: edit programs, file update programs, sort programs, and report programs. Each program type has a distinguishing structure shared by all programs of the same type. Together the four types can be coordinated in impressive data processing systems. Once beginning programmers can recognize program types, they can learn the methods used to build each type.

The logical structure for building a type of program is developed first in pseudo-code and then converted into COBOL. Example programs are carried through and enhanced from chapter to chapter as this incremental process evolves. The method of developing program logic and subsequently translating it into COBOL encourages a high degree of standardization in the programs that students produce. The similarity of their programs is the intended outcome, and the benefits of standardized program structures in the maintenance of software are continually stressed.

Use of Pseudo-Code vs. Flowcharts

I have come to believe that flowcharting and its related skills are not only mechanistic and slow, but actually create a disincentive for students to determine logical structure first and to develop the coding second. As a result, algorithmic structure is illustrated in the manuscript in pseudo-code, which is both English-like and native to our other problem-solving skills. It is a "list of things to be done." Further, with some subtle guidance, pseudo-code can take on the appearance of a skeletal program that can be translated statement by statement into COBOL.

Initial Use of Low-Volume Input/Output ACCEPT/DISPLAY

To maintain the intended emphasis on program design without getting immediately involved in the details of COBOL, it is expedient to use the low-volume ACCEPT/DISPLAY verbs for input/output (I/O) operations. Just as it is easier to teach PL/1 using the slower stream processing techniques, so it is easier to teach COBOL using ACCEPT/DISPLAY, which spares students from having to learn so much so soon about COBOL syntax. Only when students are comfortable with fundamental algorithms and their structures, with pseudo-code, and with a restricted subset of COBOL is the language fully unveiled, including the commonly used READ/WRITE verbs.

Like READ/WRITE, the ACCEPT/DISPLAY statements can be accom modated by most interactive instructional computing systems. These systems permit the user to assign logical I/O devices (e.g., SYSIN/SYSOUT, respectively) to their proper physical counterparts so that input can occur with ACCEPT and output with DISPLAY.

Comments on the Most Current COBOL Standard

Many instructors are greatly concerned that the text selected for a programming concepts course discuss the most recent accepted standard for the language under study. Undoubtedly, no textbook should omit discussion of the newer facilities a language like COBOL contains. To do so would be to equip our students with a rather incomplete knowledge of the language. By the same token, one can take an existing language standard too seriously.

Maintenance programming has become an accepted part of the information systems field. Most new programmers are exposed to the rigors of software maintenance from the onset of their employment. The COBOL programs that they will examine may contain elements of the language now discarded by the current definition. If a historical approach to teaching a programming language is not taken, these earlier instructions/statements will have little meaning to our graduates. True, they will know COBOL as it is presently defined. Unfortunately, they will have little understanding of how the language came to its present form. Keeping the foregoing in mind, I tend to include for discussion items of COBOL not presently found in the most recent version of the language. Thus, the version of COBOL presented here represents the author's own sense of what is actually being used in the industry. After all, COBOL 80 may not include a NOTE verb or a REMARKS entry, but few vendors will fail to support these items in their COBOL 80 compilers. It also includes some new facilities not presently in use but likely to be so in the near future.

Content Overview

Chapter 1 sets a reassuring mood for the benefit of the reader. It presents the business context of data processing and the programmer's role in that context as well as explaining the classification of the program types and the rationale for each. Examples are begun of payroll and inventory programs that are further developed in subsequent chapters.

Chapter 2 builds on the theme of fundamental transaction processing algorithms by focusing on the first type to be developed: report programs. These serve as our starting point because (1) almost everyone is familiar with computer-prepared reports—not so for edits, sorts, and file maintenance activities; (2) report programs assume that bad data does not exist or has been corrected; and (3) report programs can feature the least complexity of all of the algorithms introduced in chapter 1.

Chapters 3–11 use the incremental approach extensively in developing report programs. In the early chapters, the reports desired are extremely simple, without elaborate pagination schemes, lengthy title and trailer entries, or control breaks. These items are introduced chapter-by-chapter so that report programs are not concluded until chapter 11.

Chapters 12–13 introduce the algorithmic structure that uniquely identifies edit programs. Fewer chapters are required to trace the development and implementation of edit programs for several reasons. First, the beginner has already witnessed the stage-by-stage enhancement of the basic report model. Second, much less of COBOL is required to reach the instruction stage of edit programs. Third, some portions or segments of the two program types can be used in both.

Chapters 14–15 provide the more thorough coverage of the IDENTIFICATION and ENVIRONMENT DIVISIONS that is appropriate at this point.

Chapter 16 discusses the definition and manipulation of conventional files. It also finishes the expanded coverage of the ENVIRONMENT DIVISION by dealing with the INPUT-OUTPUT SECTION, and discusses the disadvantages associated with ACCEPT/DISPLAY in preparation for replacing them with READ/WRITE.

Chapters 17–18 present a thorough discussion of table definition and processing.

Chapter 19 emphasizes the importance of modular design for software maintenance and discusses external module definition and use.

Pace

The incremental approach followed in developing the chapter content is geared to the beginning programmer. Classroom experience has convinced me that the topical development moves at an appropriate pace for the beginner, though it may appear slow to instructors, who are well versed in COBOL.

This text contains more material than can normally be thoroughly covered in a single term. In a first programming course, it is likely that students will reach and perhaps complete the first chapter on table definition and processing. Students who have studied another programming language will move more rapidly through the initial chapters and probably reach the last chapter on external module definition and use.

Pedagogical Features

Each chapter includes learning aids that facilitate mastery of the content:

Chapter Outlines give an overview of chapter content for easy reference and a sense of the chapter's general aim.

Running Glossary in the margins of each chapter highlights important terms and their definitions as they appear in the text discussion.

Statement Formats, located in appropriate places within chapters, provide students with the general formats for COBOL entries. These statements are gathered together in appendix C for easy reference.

Master Glossary gathers all important terms and their definitions in a single alphabetical list.

Chapter Summaries provide a concise review of the content of each chapter that clarifies the essentials.

Questions for Review challenge the students to synthesize the concepts presented.

Problems provide the practice students need to master the COBOL syntax currently under study. Students will find these and the Questions for Review invaluable as preparations for programming assignments and exams.

Programming Assignments, which begin following chapter 3, ask students to use the skills they've learned to write programs from the context provided. The data to be used in these programming assignments is included in the accompanying Instructor's Manual.

The text also includes three appendixes that (1) review the steps in constructing COBOL programs from the perspective of antibugging and debugging techniques, (2) list the reserved words in COBOL, and (3) list the general formats for COBOL coding entries.

Text Supplements

Introductory Structured COBOL is accompanied by two supplementary books.

Instructor's Manual

An Instructor's Manual is available that contains

Chapter overviews and teaching suggestions
Important terms and their definitions
Answers to review questions
Answers to problems
Solutions to programming assignments, including input data
Test questions and answers

Casebook

The casebook provides students with additional experience in constructing COBOL programs. It presents two cases per chapter that parallel the material presented in the text and are similar to the programming assignments in the text. One programming project is worked through in detail, including material on antibugging and debugging. The programs developed as solutions to the case problems demonstrate the proper approach to the programming assignments in the text. The second case for each chapter is presented as an additional programming assignment for the student to work out.

Solutions to these assignments (programs and input data) as well as the solutions from the Instructor's Manual for programming assignments in the text are available on magnetic tape. Interested adopters may send a blank tape (include return postage) directly to the author at The American University, Dept. of Mathematics, Statistics, and Computer Science, 221 Clark Hall, Massachusetts and Nebraska Ave., Washington, DC 20016.

Acknowledgments

The following information is reprinted from *COBOL Edition 1965,* published by the Conference on Data Systems Languages (CODASYL) and printed by the U.S. Government Printing Office.

> "Any organization interested in reproducing the COBOL report and specifications in whole or in part, using ideas taken from this report as the basis for an instruction manual or for any other purpose is free to do so. However, all such organizations are requested to reproduce this section as part of the introduction to the document. Those using a short passage, as in a book review, are requested to mention "COBOL" in acknowledgment of the source, but need not quote this entire section.
>
> "COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.
>
> "No warranty, expressed or implied, is made by any contributor or by the COBOL Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection therewith.
>
> "Procedures have been established for the maintenance of COBOL. Inquiries concerning the procedures for proposing changes should be directed to the Executive Committee of the Conference on Data Systems Languages.
>
> "The authors and copyright holders of the copyrighted material used herein
>
> > FLOW-MATIC (Trademark of Sperry Rand Corporation) Programming for the Univac (R) I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation; IBM Commercial Translator Form No. F28–8013, copyrighted 1959 by IBM; FACT, DSI 27A52602760, copyrighted 1960 by Minneapolis-Honeywell
>
> have specifically authorized the use of this material in whole or in part, in the COBOL specifications. Such authorization extends to the reproduction and use of COBOL specifications in programming manuals of similar publications."

I want to gratefully acknowledge the contributions of those who reviewed all or portions of the manuscript for this text and who supplied detailed criticisms and valuable ideas. These people include Joyce L. Abler, Central Michigan University; Norman Brammer, Colorado State University; Martin H. Cronlund, Anne Arundel Community College; Caroline N. Curtis, Loraine County Community College; Chester Luck, Thomas Nelson Community College; Malik M. Nazir, Western Michigan University; John F. Schrage, Southern Illinois University, Edwardsville; Dennis Varin, Southern Oregon State College; and John Werth, University of Nevada, Las Vegas. Most authors feel compelled to both express their gratitude and, at the same time, exonerate those whose assistance has proven invaluable in their respective undertakings. A book such as this one makes such action imperative.

There is one individual who stands apart from the rest. At the risk of sounding maudlin, Michael Greene has set an intellectual pace that is both exhilirating and addictive. Finally there is my bride who has sacrificed more than she will ever receive as compensation.

Introductory Structured COBOL

Introduction

1

Chapter Outline

Planning: The Basis for Future Action

We are all schemers to a certain degree. Who hasn't at least thought of owning their own company? Of being their own boss? Each scheme represents a plan for carving out one's own niche in life. Organizations are much like individuals—they too have plans. These plans, however, are the end products of the work of many people within the organization. A manufacturing concern, for example, must identify those products that will be enthusiastically received in the marketplace if the company wants to remain profitable. Indeed, just identifying the products that consumers desire is not always enough since other organizations might market a very similar product. Competition, then, forces many organizations to identify not only saleable products, but also to research the best means of packaging, transporting, and advertising their products in order to gain a competitive edge.

Organizing: The Acquisition and Allocation of Resources

Once a plan of action has been formulated, the managers of an organization must begin acquiring and allocating the resources needed to bring the plan to fruition.

Are there sufficient raw materials with which to build the product? Where can the materials be purchased? Does the company have the expertise required to manufacture the product? The number of questions to be answered seems endless.

Controlling: Are We Proceeding According to Plan?

Every plan needs a measurement system. The manufacturing organization requires a quality assurance function in order to determine if the product appearing at the end of the assembly line is of the desired quality or workmanship. The company must also monitor incoming orders so that production schedules reflect demand. Frequently, the measurement system will yield some very unsettling information; that is, that all is not proceeding according to the plan. The nature of the organization's reaction to such disturbing information will determine its success in the marketplace. Failing to respond promptly could lead to disaster.

Information: The Passport to Intelligence

It should be evident that large organizations require timely information if their control system is to function properly. Without timely and accurate information, it is unlikely that any firm can properly determine whether or not it is proceeding according to plan. Consequently, most organizations have very elaborate data collection mechanisms that help monitor the vital signs of the company. Not all of these data collection techniques are new. Accounting is an excellent example of a system that existed for years.

Accounting systems gather financial data that are formatted for reports to managers. These reports depict the organization's financial health. For instance, accounts payable reports reflect corporate debt. Accounts receivable reports indicate who owes money to the company. These and other financial reports form a comprehensive view of an organization's financial affairs and are the object of constant scrutiny by management. They are part of the measurement system that notes deviation from established goals and objectives.

Automation: Coping with a Wealth of Data

The difficulty of collecting data for the measurement system varies greatly. Smaller organizations, whose operations are geographically centralized, tend to have fewer problems collecting data than do their decentralized, larger counterparts. The independently owned corner grocery store is an excellent example of the small organization. The proprietor works full time in the store stocking shelves, placing orders with suppliers, tending the cash register, and greeting customers. This intimacy permits the owner to recognize trends, to minimize inventory, and to manage the financial functions.

Larger organizations, however, must operate in an altogether different environment. These organizations often employ a large number of people to prepare accounting reports, to manage inventory, and so on. The personal versatility encountered in the "Mom and Pop" grocery store is just not possible in such a large organization. A large chain of grocery stores must monitor and maintain the inventories for each of its stores. Hundreds, perhaps thousands, of employees must be paid on a regular basis with complete accounting of sick-leave and vacation balances. In addition, sales trends must be anticipated for the various regions that the company serves. The requirements of data gathering, report preparation, and report distribution in an accurate and timely fashion has forced many organizations to consider *computer-based measurement systems*. There is just too much data to be collected and reported for this task to be accomplished manually.

Computer-based Measurement Systems
The collection of computer-produced reports that indicate whether an organization's plans are being fulfilled.

The Players

The computer is a tool of management. The sole product of computer-based measurement systems is the printed report. These reports vary tremendously in terms of their content as well as how often they are produced. The individual who designs these reports is the *information systems analyst*. Like the computer, the information systems analyst should be viewed as a tool to be used by other managers.

Information Systems Analyst
The individual who designs computer-generated reports and the programs that produce those reports.

Managers are responsible for making the best policy decisions with regard to planning, organizing, and controlling. The best plan is the one that takes the organization into the most profitable markets and results in a stable level of financial resources. The proper organization and allocation of resources will guarantee that the best product possible is being produced at the least possible cost. Effective controls assure that the desired marketplace is being approached as planned and in the expected manner.

Policy decisions are not at all easy to execute even if they are properly done. Indeed, each of these decisions is the outcome of a choice of alternative strategies. Each strategy has consequences. The most desirable strategy is the one that is most likely to produce the best result. To choose among alternatives, a manager needs relevant, accurate, and timely information. It is the information systems analyst's job to help managers determine what information they may require to assist their decision-making processes.

In many respects, the information systems analyst is much like the architect who assists in determining the general style and layout of a dream home.

The architect's artistic rendering is a model of a home that has yet to be built. Similarly, the information systems analyst presents the corporate manager with a model, or facsimile, of the printed report. Like the architect's final renderings, which are a product of many sessions with the prospective homeowner, the facsimile reports are not likely to be in final form after just one meeting between the analyst and the manager. The manager must first come to grips with a basic but difficult set of problems. What reports are needed? What must be the content and format of each report? What data gathering procedures are required?

The analogy does not end here. When the general design of a home is finished, the architect must translate that model into a blueprint for building. Once the blueprints have been completed, the actual process of building can begin.

The information systems analyst seeks the approval of his client manager just as the homeowner must approve the architect's rendering of the future home. The manager is asked to approve a sample of a report that will be computer produced. Approval may not be immediate since the manager may not be satisfied with either the information that appears on the sample report or its format. As a result, the analyst must revise the report until it is approved.

Like the architect who produces a blueprint, so must the information systems analyst produce a blueprint of sorts. This blueprint will be used to build, and then to make operational, a series of computer programs. These computer programs will produce an exact replica of the report approved by the corporate manager. Some reports cannot be computer produced, and not all of those that can be, should be. The reasons for this phenomenon vary.

Reports do not just magically appear from computer programs. The task of a computer program is to transform data, or inputs as they are known, into an exact replica of the report that has been requested by a corporate manager. Not all of these inputs already reside within the computer. In fact, just the opposite is true, so seldom are desired data elements computer resident that the information systems analyst must include a data gathering process in the blueprint. Some data will prove too expensive to be collected. Reports from a computer-based measurement system are supposed to help note corporate problems and thus save the company significant sums of money. If producing a certain report by computer costs more than that report can save, it is unwise to invest corporate funds for this purpose. However, many computer-based measurement systems are cost-effective and many are actually constructed.

Computer Programmer
The individual who constructs computer programs in a programming language, such as COBOL.

The blueprints developed by the information systems analyst are assigned to one or more *computer programmers*. The number of programmers needed depends upon the number and complexity of programs to be built.

Computer programmers are guided in their task by two items. First is the facsimile of the report that was approved by the manager. Since the computer programs must replicate the facsimile, it serves to verify that the format produced by the computer is correct. The second item is the blueprints for computer program design. These blueprints instruct the programmer in the nature and format of the inputs and in any special formulae that are to be used.

The computer programmer must write the programs that will produce the reports. These programs are a collection of data definitions and instructions for manipulating these data written in one of the popular programming languages: FORTRAN, PL/I, RPG, and COBOL.

COBOL
(COmmon Business Oriented Language) A language in which a computer programmer may build computer programs for business applications.

COBOL, an acronym for COmmon Business Oriented Language is the focus of this text. This language is by no means the only one for computer-based measurement systems. It is, however, one of the most popular programming languages currently used in business applications. This text will illustrate how computer-based measurement systems can be constructed using the COBOL programming language.

Point of Departure

So, you want to be a COBOL programmer? You know that the ultimate client in any information systems project is a corporate manager. Furthermore, you should also understand, in a very general sense, the role of the information systems analyst. Now, where do you fit in? What do you do? To better understand your role in this entire process, it is a good idea to briefly review what has already taken place.

Your client has recognized a need for more information to effectively manage one or more areas of responsibility. Consequently, he or she requests the assistance of the information systems analyst. Following much discussion, the two individuals are able to determine which reports are needed. These reports are the facsimiles of those that will be produced by the computer. Once the client's information system has been further defined by both the analyst and the programmer, the programmer builds, tests, and documents all the programs and data files that constitute the information system.

The clue to your exact role is found in the last statement. The analyst and the programmer further define the information system, and the programmer does the rest. The programmer builds, tests, and documents all of the programs and data files that comprise the new information system. In your role as the programmer, you will construct and test the program that will manipulate files of data to yield a desired report.

Just what is implied in the phrase *further define?* First, the facsimile of the output report must be examined. Reports contain information. Each item of information must be identified. Next, all of the input data items required to produce those items of information on the output report must be identified as well. An example of a payroll report will further explain the nature of this process.

Assume that a manager of corporate payroll operations needs a report containing the following information for each of the employees in the company: (1) employee name, (2) social security number, (3) hourly wage, and (4) the number of hours worked for the current pay period. The report might look similar to the one shown in figure 1.1.

Together with the analyst, you as the programmer would identify the items of output data that appear on the report shown in figure 1.1. These items must be generated by the computer program that will produce this report. While this report contains many items, one in particular is of special concern here: hours worked. Where does this item of data originate? It is likely to come from the time cards completed by each employee at the end of the pay period. What about the other items of output data? Where do they originate?

Time cards can only hold so much data. In fact, it is difficult to obtain from them much more than the number of hours worked each day during the pay period. As a result, this computer program needs two separate and distinct data files to process as inputs. One is, of course, a file containing nothing more than a time card for each employee who is to receive a paycheck. Second is a file which is biographical in nature.

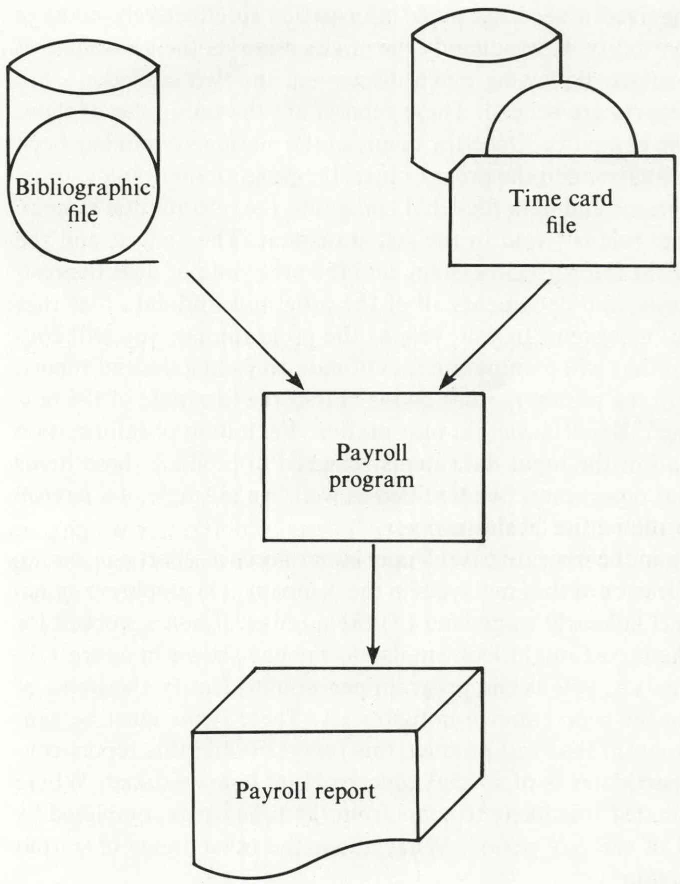
Figure 1.1
Sample payroll report

NYC CENTER FOR DISEASE PREVENTION
AND CONTROL

WEEKLY PAY REPORT

SSN	NAME	HOURS	RATE	GROSS
215-50-3145	WEINER, HD	12	1.85	$22.20
197-42-1508	BAGGINS, FX	15	3.21	$48.15
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

Figure 1.2
System flowchart for report program



This second file needs to contain data such as the name of the employee, the employee's hourly wage, etc. The only item that would most likely appear in both files is the employee's social security number. This common datum permits a computer program to match the entries of the two files for the purpose of producing the facsimile payroll report. A match exists when records from each file contain the same social security number.

The only item of data not found in either the biographical file or the time card file is each employee's gross pay for the pay period. The reason is quite simple. Gross pay will be computed by the computer program. After all, if gross pay is on the time card, it must have already been computed by someone or something, making the program unnecessary.

Any item that appears on the report can be described as being present in an input file, or as being computed by the computer program based on data that are present in an input file. In this case, gross pay is computed from the hours worked shown on the time card and the employee's hourly wage from the bibliographic file.

System Flowcharts
Graphic illustrations that name the input files used by a computer program as well as the output files produced by that program.

Based on this example, the analyst and the programmer can produce a *system flowchart* such as that shown in figure 1.2. This flowchart describes the basic data flows necessary to produce the payroll report. It names both input files and gives the name of the report. Yet this alone is not enough information with which to proceed. It is not sufficient to know that this program will have two input files and produce one report. As a programmer, you must also know how the data in those files are organized. For example, in the biographical file, where can the employee's name or hourly wage be found? To complete the program design description, you must also specify the size of the records in each input file and the position of each item of data within every record.

Record Layouts
Layouts that describe the names, attributes, and physical location of the fields that constitute a record.

An example of the *record layout* for the input file that contains time card images is shown in figure 1.3. Similar record descriptions can be specified for any input file and for printed reports as well.

File and record layout for time card file

Card columns	Type	Data
01–11	Alphanumeric	Social security number
12–19	N/A	Unused
20–32	Alphabetic	Employee's name
33–39	N/A	Unused
40–41	Numeric	Hours worked
42–49	N/A	Unused
50–53	Numeric	Hourly wage
54–80	N/A	Unused

Figure 1.3
File and record layout

Programmer's Role

As a COBOL programmer, you are expected to assist the information systems analyst in decomposing facsimile reports into a system flowcharts. These flowcharts are only possible after you have examined the facsimile report. You must determine which data items can be found directly in the input and how many input files are necessary. Having done that, the required record descriptions can be constructed. In addition, you will be expected to translate system flowcharts, record layouts, and facsimile reports into a working COBOL program.

Where to Start

The system flowcharts define the inputs to the program; the facsimile report tells you what the program must produce; and the record layouts for each file help you locate the data. What do you do next?

As you know, programmers write computer programs that produce outputs when given certain inputs. COBOL programmers write these programs in the COBOL language. This you also know. Yet something seems to be missing. How do you actually translate the documentation just described into working programs written in COBOL?

Programming is a two-step process. The first, and most important, is designing the program's logic. A program's logic is the general rules and processes that the computer program must successfully follow if the desired outputs are to be produced. To write a computer program in any language without properly determining its logic beforehand is *always* a futile exercise.

Programming
A two-step process in which the programmer first determines the actions a computer must take to transform inputs to outputs, and second, translates those steps into a language such as COBOL.

Poor Programming Technique
Beginning programmers who want to start step two of the programming process before step one is completed.

Good Programming Technique
Beginning programmers who do not progress to step two of the programming process until they have first triumphed in completing step one.

The second step involves translating these logical operations into a programming language—in this case COBOL. This task can be challenging as one attempts to find the best translation possible. Best, in this sense, may mean using the least possible amount of computer memory or the least amount of processor time. Ease of reading and of interpreting a program's logic when reviewed by another programmer is also important.

Translation is the easier of the two steps. Beginning programmers tend to start writing the programming instructions using a programming language as soon as they possibly can. They start the second step before the first is completed. It is easy to be misled into believing that the real challenge in computer programming is speaking a programming language, like COBOL. This simply is not true.

The very first task that any programmer must confront is determining a program's logic—how it will process the input data to produce the desired outputs. Only after the program logic has been fully developed should the programmer convert these processing rules into COBOL.

Why do programmers fail to follow the steps just outlined? The first step in programming, that of logic design, is a thought-intensive, physically passive task. Translating the logic of the program to COBOL, the second step, involves a much greater level of physical activity. Apparently, many programmers confuse activity with progress. Translating incomplete and ill-conceived logic into any programming language serves only to increase the time required to finish a programming assignment. Furthermore, even if a working program is produced under these circumstances, its logical structure will more closely resemble a rickety house than a thing of beauty. The problem in programming is not the mastery of a particular language. The problem is not COBOL, but is, rather, the programmer who starts the coding of solutions before those solutions are known.

Do not worry about mastering COBOL at the present time, because you will develop that skill soon enough. Instead, be absolutely obsessed with learning how to design the logic of programs so that inputs can be transformed into the outputs specified.

Hopefully, you are now looking for some wisdom in attacking the first step of the programming process. How do you design a program's logic? You will soon find that design of program logic is not as overwhelming as you might think.

Program Classification

Any computer program can be classified as: (1) an *edit program,* (2) a *file update program,* (3) a *sort program,* or (4) a *report program.* What does this mean to you? This is a guide to use in determining the basic logical structure of the programs you desire to construct. Note the following example.

The inventory manager of a sales organization has requested the assistance of the information processing department. This new client is responsible for maintaining an adequate inventory of the thousands of items carried in the corporate inventory. Having too few of a given item means that customers will go elsewhere to buy the product. Having too many means that the company has overinvested its scarce capital in unnecessary quantities of a product. The information systems analyst helps the inventory manager devise a report that will show the difference between the quantity of items in stock and the quantity desired by the customers. An example of such a report is illustrated in figure 1.4.

This report, when produced on a timely basis, allows the inventory manager to order products in short supply and avoid ordering additional, unnecessary quantities of products.

The analyst, when meeting with the programmer assigned to this project, notes the following. In order to update the inventory, one or more computer programs must process sales slips and inventory ordering forms. The sales slips are created whenever a customer purchases one or more items from the company. Sales slips contain the stock number and quantity of the items purchased. The inventory ordering forms are completed by the management staff when an item must be replenished. Since both of these inputs are presently handwritten, the data contained on them need to be converted to a form that can be processed by the computer.

Figure 1.4
Inventory report

STATUS OF CORPORATE
INVENTORY

STOCK NBR	DESCRIPTION OF ITEM	QTY ON HAND	QTY NEEDED
134759	PENCILS	10	190
134761	ERASERS	1,115	5
.	.	.	.
.	.	.	.
.	.	.	.

Handwritten documents, except in special circumstances, cannot be processed directly by the computer. Consequently, the contents of these sales invoices and inventory order forms must be transferred to a computer medium such as the punched card, magnetic tape, or magnetic disk by a key operator. The key operator reads the information from each document and then keys that information, one character at a time, onto the chosen medium.

Edit Programs

Two problems arise in this transcription process. First, the key operator may incorrectly key the data onto the new medium. Second, the employee who originated the sales slip or inventory ordering form may have erred when entering information on that form. As a result, it is generally assumed that data contain errors. Even though a partiuclar sales slip or order form may be correctly written and transcribed, it must still be checked for accuracy.

The purpose of an *edit program* is to check input data. Input transactions (source documents are also called transactions) containing errors must be resubmitted with the appropriate corrections.

Edit Programs
Programs that check the accuracy of data that has been transcribed from source documents to machine-readable form.

Usually, large numbers of input transactions are gathered in *batches* to be edited as a group. Batched input saves computer costs and improves the procedures that computer support personnel must follow. Input transactions might be held in a batch until the end of each week before processing. Thus the edit progam, which checks the accuracy of inventory transactions, might only be executed once a week.

Batch
The transactions that have been collected over a period of time for processing as a group.

The edit program should produce its own report (not to be confused with the report desired by management) that lists each incorrect transaction and the nature of each error. The mistakes can then be correctly repunched or rekeyed.

Update Programs

Many computer-based inventory systems maintain a file containing representative entries for each product. These entries are called inventory records. Each product has its own record in the inventory file. These records contain the identifying number of the product, as well as other information that reveals the status of that particular item. The information might include

1. stock number assigned to the item
2. description of the item
3. unit prices (acquisition and resale)
4. quantity currently on hand
5. quantity purchased (acquired and sold)
6. date of last order
7. quantity ordered but not yet received

If 5,000 products are in the inventory, the inventory file would contain an equal number of records.

Update Programs
Programs that allow new records to be added, and existing records to be altered or deleted from files of related records. (Also known as file maintenance programs.)

An organization must frequently update the status of the inventory file to reflect sales of stock. This is done by applying the sales invoices to the inventory file. When a match exists between a sales invoice and a record from the inventory file (i.e., they both have the same stock number), then the inventory record is updated to reflect the loss of the product sold.

Sort Programs

The entire concept of file maintenance requires that the sales invoice transactions be sequenced by stock number in ascending order (lowest stock number to highest stock number.) The proper sequencing of the input transactions must be done in one of two ways. Either the input transactions are ordered manually by the key operators at the time the data is transcribed from the original source documents, or the file of input transactions is ordered by the computer prior to running the update program. Of the two alternatives, the second is the more attractive. The first alternative assumes that the key operators are immune to error and will not missequence the input transactions. The second alternative, the computer sort program, works properly every time it is used.

Sort Programs
Programs that sequence, or order, records in a file in a manner specified by the user of the program, usually provided by the vendor of the computer system.

The *sort program* is generally provided by the vendor who supplies the computer hardware and is successfully used by a large number of computing installations. The program is totally accurate; consequently, seldom, if ever, will a programmer be called upon to design and construct a sort program.

Report Programs

Report Programs
Programs that read, reformat, and then print selected records from an input file. Typically, the format of the printed report has been specified by the manager/client.

The *report program* produces the report required by management as part of their measurement system. This report, for example, divulges the status of the inventory. Remember, a properly maintained inventory is the goal of the inventory manager.

Common Structure

You might wonder what the advantage is in classifying all programs into these basic types. Its virtue is that, for instance, all report programs have a strikingly similar logical structure. So, once you become familiar with the structure of report programs in general, you can draw upon this knowledge when assigned to design and construct your specific report program. You know where to begin when given a report programming task. The same is true of the other program types.

Chapters 2 through 11 are dedicated to developing the general structure of all report programs. Because this program type is developed over the next ten chapters, you will see a report program *model* evolve from a very simple listing of an input file to one that embraces titles, trailers, summary totals, and finally control breaks.

Model
A guide for building a computer program in which the actions of that program are already known. For example, edit program models differ from report program models due to their different function or purpose.

Chapters 12 and 13 introduce the general logical structure of edit programs. Fewer chapters are required for the development of this model because the process has been experienced once already.

Learn how these models come to be what they are. Recognize that memorizing the specifics of each model will provide you with a powerful tool in the design of program logic. Knowing where to start and with what to start will greatly ease program design for the beginner.

It should be noted that the particular classification scheme described in this chapter is not the only one possible. Indeed, many exist. For the purpose of this text, however, we shall assume that edits, updates, sorts, and report programs describe every possible computer program that can be constructed. What is important to realize is that similar groupings can be created. The general logical structure of each type of program can be developed. The program models are absolutely critical to the programming process; with them programmers know where to begin their logic designs.

Summary

Any computer-based measurement system requires files of data in order to produce the reports desired by management. Unless these files are already resident in the computer, they must be built and maintained. To accomplish the maintenance function, procedures must be established to collect data from source documents reflecting basic organization transactions. These documents are frequently handwritten, and thus cannot be processed directly by the computer. As a result, key operators must transcribe these transactions onto a machine-readable medium, which in turn is read by the computer and edited for accuracy. Once a batch of transactions has been edited, the file can be sorted by a key field, such as stock number, in preparation for a file update process. After the file has been updated to reflect the previous week's inventory activity, the report program can produce a status listing of each inventory product.

Thus every program that can be written by a computer programmer, in any language, must fall into one of the following four categories.

1. Edit programs—identify errors introduced when data were recorded or transcribed into machine-readable form.
2. Sort programs—resequence input transactions to permit file updates.
3. File update programs—modify the contents of a file by adding new records, deleting existing records, or modifying existing records.
4. Report programs—produce the printed report that the entire data processing system is designed to yield. The report to be used by the manager in decision-making activities.

Several items are worth noting. First, only one program actually produces the report requested by management. This is the report program, the last of the four kinds of programs just described. Yet to reach that point, data must be collected, edited, sorted, and the master files must be updated. Only then can the manager's report be produced.

Second, any program that you, the programmer, can write must fall into one of three categories. The program is an edit, an update, or a report program. Sort programs do not need to be written because a general purpose sort program is provided by the vendor who supplies the computer system.

Finally, we can develop the general logical structure for each program type within this classification scheme. Knowledge of these general structures, or models, provides a convenient starting point in the design of program logic. First, we determine whether our program is an edit, update, or report. Second, we locate or recall the general structure of the appropriate program type. Third, we customize the general model to encompass the programming assignment we have been given.

Questions for Review

1. Why is the information systems analyst much like an architect who helps a homeowner design a dream home?
2. If the analyst is much like the architect, then who is the programmer like in terms of home design/construction?
3. Why must the analyst examine each item of information that is to appear in the facsimile report?
4. Produce a possible record layout that would describe the report entitled *Status of Corporate Inventory*. Assume that each printed line on the report is eighty characters in width.
5. Examine the following sequence of events in the processing of inventory transactions.
   a. sale occurs and is recorded on handwritten invoice
   b. data on invoice is transcribed by key operator onto machine-compatible medium
   c. inventory file is modified to reflect sale

   Why is it unlikely that these steps could actually occur in an automated data processing environment? (Hint: What steps have been omitted?)
6. Examine the following sequence of computer programs.
   a. edit program challenges validity of data in transactions
   b. file maintenance program updates contents of master file based upon data from step (a)
   c. sort program resequences transactions
   d. report program produces listing of the contents of the master file following modification in step (b)

   Is this sequence correct?
7. What is the advantage of a program classification scheme like that offered in this chapter?
8. What is the danger, if any, in beginning to code before a solution has been completely designed?

The Structure of Report Programs

2

Chapter Outline

The report program is the first of the four types of computer program we will examine for its logical structure. This may appear to be an unusual place to begin. After all, the report program was the last of the four to be described in chapter 1. It is, however, a very appropriate starting point. Report programs can be understood with the least exposure to COBOL. The fewer statements to master at the outset the better.

Report programs produce a written report as their *principal product.* Files are not updated, records are not resequenced, and input transactions are not edited during the execution of a report program. The only thing a report program is expected to do is to produce a desired report. This report must contain the necessary information to assist management in the decisions it must make. It is assumed that the data the report program uses is ready to be processed and free of errors.

All report programs have the same basic logical structure no matter what report is being produced. Different reports for different applications (e.g., accounting, personnel, etc.) have unique report titles. And the data appearing on each report are unique to that report. However, the processing structure of every report program is the same. It doesn't matter whether the report is being produced for the payroll manager, the personnel manager, or even the inventory manager. All of these programs have a strikingly similar basic structure.

When you learn the nature of the report program's structure, you have substantially mastered the hardest step in the programming process. If someone requests that you design a report program, such as for a payroll application, you will know how to begin. First, you will recall that all report programs have the same structure. Second, you will start with the skeletal structure of the report program logic and add the appropriate report titles and descriptions of the data being reported. Remember, it is the appearance of each report that makes it seem unique. Each report program has the same skeleton. The titles and data descriptions represent the flesh, muscle, and tendons that make reports appear to be different despite the similarity of their skeletal structure. Third, you will translate the model complete with titles and data descriptions into COBOL. You will use this procedure for building report programs again and again.

Iterative Nature

Iterative
An activity that occurs repeatedly for a fixed number of occurrences.

Report programs can be characterized as *iterative* in their approach to processing data. That is, the computer continues to produce the desired report line-by-line until no more input data records remain to be processed. Each record is read, one at a time, from the input file with each input record corresponding, more or less, to one line on the printed report. In the case of the inventory report noted earlier, if 100,000 items were in stock, then a status report would contain 100,000 entries.[1]

Each entry appearing in the printed report shown in figure 2.1 is extracted from the inventory file and printed one at a time. It is clear from this report that our inventory file contains many records. One record is for pencils and another is for erasers. The printing process begins with the first record from that file, proceeds to the next, continues to the next, and so on, until the file is exhausted of all records. Each input record extracted from the inventory master file and transferred to the report corresponds to the record description shown in figure 2.2.

Pseudo-Code

Computer programming has been described thus far as a two-stage process. The first stage might be termed program design, while the second might be termed translation of the design into the statements of a programming language. How do we express or illustrate a program design? One possibility is to use a programming language. Unfortunately, this alternative requires that you learn a programming language before you master program design. This makes very little sense. Besides, if we write our program designs using a programming language, we are engaged in the second step of the programming process before the first step is concluded. This is much like trying to

Figure 2.1
Prototypical report program

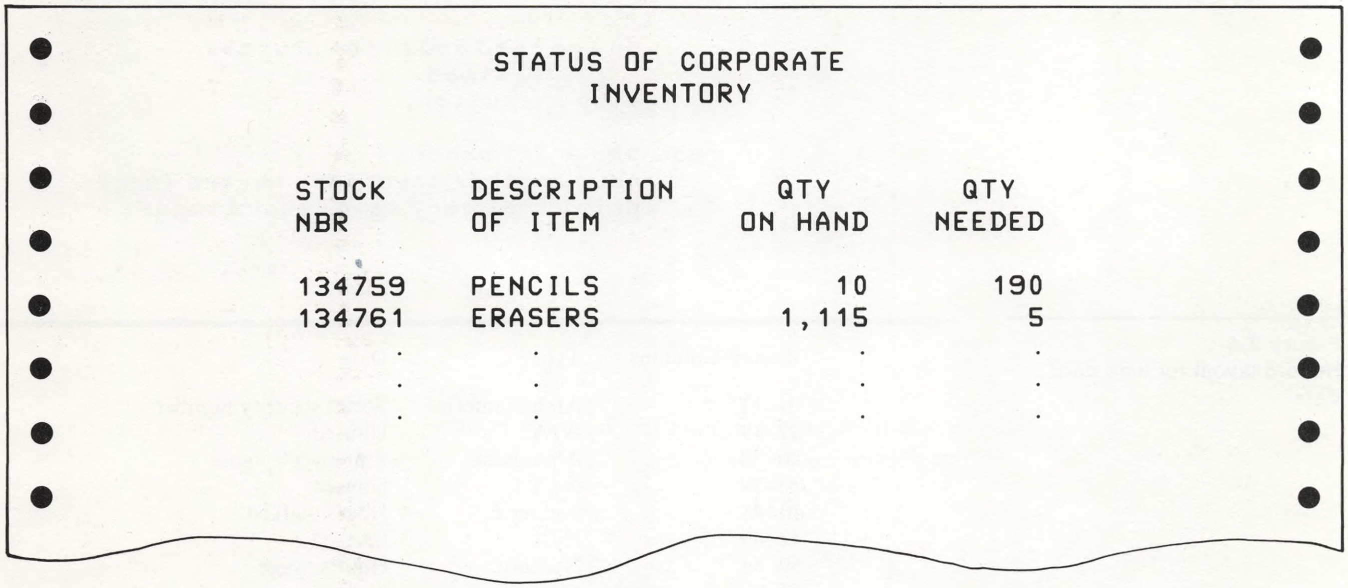


Figure 2.2
Description of records in inventory file

File and record layout for inventory master file

Card columns	Type	Data
01–06	Numeric	Part identification number
07–19	N/A	Unused
20–34	Alphanumeric	Part description
35–40	N/A	Unused
41–43	Numeric	Quantity currently in stock
44–54	N/A	Unused
55–57	Numeric	Quantity required if all pending orders are to be filled
58–80	N/A	Unused

make adobe bricks without straw. The straw holds the mud together when dried. In a similar manner, a program's logical structure ties the individual programming statements together in a meaningful fashion.

A better way to illustrate a program design is to use a symbolic language that can adequately describe the functions within an existing program. This symbolic language must be capable of describing the actions of all types of programs—edit, sort, file update, and report programs—encountered in business data processing. Further, it must be relatively easier to write than the statements in the COBOL programming language. Such an informal language exists in the data processing field and is known as *pseudo-coding*. Pseudo-coding looks like a programming language, but it isn't. It is easy to learn. The statements used are short, action-oriented phrases. Control statements in pseudo-coding consist of three types. They are:

Pseudo-code
A shorthand, English-like language in which program logic can be expressed.

1. Sequential structure—symbolizes an input operation (read a record) from a file of data. It can also be a simple arithmetic instruction.
2. Decision structure—determines if a desired event or set of circumstances has yet occurred.
3. Iterative (looping or repeating) structure—controls the number of times we choose to repeat a series of events.

Figure 2.3
Program logic expressed in pseudo-code

Statement number	Pseudo-coded statement
1	do the read and print process 10 times.
2	stop execution.
3	read and print process
4	read another inventory record image
5	print that very same record image.

Figure 2.4
Record layout for time card data

Record Columns	Type	Data
01–11	Alphanumeric	Social security number
12–19	N/A	Unused
20–32	Alphabetic	Employee's name
33–39	N/A	Unused
40–41	Numeric	Hours worked
42–49	N/A	Unused
50–53	Numeric	Hourly wage
54–80	N/A	Unused

80–80 Listing
A report program that prints each 80 column record image read from the input file without reformatting the contents of the record.

Sequential Structure
An instruction that does not cause the flow of program control to change from its serial nature.

The best way to illustrate the use of pseudo-code is to structure a pseudo-coded version of the inventory report program noted earlier.

The report program logic illustrated in the pseudo-coding can be better understood if it is traced instruction by instruction. Figure 2.3 is an example of the skeletal structure of an *80–80 listing* report program. An 80–80 listing is a reproduction of the input record images upon the printed report with no alterations whatsoever. The data contained in each input record is printed exactly as it is read.

Statements 2, 4, and 5 are examples of *sequential* pseudo-code control structures. After the pseudo-code is translated into COBOL, the computer executes each of these instructions. The action specified is carried out, and control in the program proceeds to the very next pseudo-coded instruction. The only exception to this rule is statement 2. Statement 2 causes the program to come to a complete halt. Since execution is halted as a result of executing statement 2, control is not automatically passed to the next instruction (statement 4).

Statement 3 is not actually a pseudo-coded instruction. In fact, this statement is not an instruction at all. This particular line serves as a documentation aid required by statement 1. Thus statement 3 helps identify which of the statements in our program belong to a function known as the *read-and-print-process.*

Statement 1 can be thought of as an iterative structure. An iterative structure permits the execution of a group of other statements a specified number of times. The iterative structure labelled statement 1 notes that the group of instructions to be repeatedly executed is known collectively as read-and-print-process. Thus statement 3 serves as a group statement identifier. That is, statements 4 and 5 belong to the group named (identified) in statement 3.

The sample pseudo-coded 80–80 listing program does not yet contain an example of a decision structure. That will be added shortly.

Assume that this program logic is to reproduce the record images described as time cards in chapter 1. Figure 2.4 represents the file and record layout for the input data.

The report produced by the 80–80 listing program logic could appear as shown in figure 2.5.

Figure 2.5
Output produced by 80-80 listing

```
          .              .            .
          .              .            .
          .              .            .
    215-50-3145     WEINER        34     1.99
    197-42-1508     BAGGINS       18     6.44
          .              .            .
          .              .            .
          .              .            .
    177-55-8765     KASIMIRSKY    25     3.90
```

Figure 2.6
Program logic expressed in pseudo-code

```
Statement        Pseudo-coded statement
 number

   1                 do the read and print process
                           10 times.
   2                 stop execution.

   3           read and print process
   4                 read another inventory record image
   5                 print that very same record image.
```

Note that the descriptions of the input data and the output data, as they appear on the report, are exactly alike. Nothing has been changed or reformatted. Each line in the report contains a social security number followed by a series of eight blanks, followed by the employee's name, followed by a series of seven blanks, and so on. Each line is an exact replica of the input data. Also, notice that this report does not contain many of the refinements managers expect to be present on reports, such as report titles, date of preparation, page numbers, and the like. We shall concern ourselves about these particular items at a later time.

Inner Workings of the Iterative Structure

The program logic reproduced in figure 2.6 is composed of two parts. The first portion or section in figure 2.6 is represented by statements 1 and 2. This section may be thought of as the main control portion of the entire program. Indeed, statement 1 controls the entire read-and-print-process that this report program is to perform. The second section of the program is the actual read-and-print-process.

It is worth discussing just how the iterative structure in statement 1 actually works. When executed by the computer, control is automatically transferred to the first of the statements in the group identified as the *read-and-print-process.* That is, control goes immediately from statement 1 to statement 4. Statement 2 is passed over, or skipped. Once statement 4 completes its function (i.e., the next record image is read), then control proceeds to statement 5 (i.e., the record image just read is printed). After this last statement is completed, then control returns to statement 1.

Beginning programmers are often perplexed about how and why control automatically proceeds from statement 5 to statement 1. The reason is really straightforward. Iterative statements are designed to cause one or more other statements to be repeatedly executed a set number of times. In this instance, statements 4 and 5 constitute the group to be repeatedly executed. Control must automatically return to statement 1 after statement 5 to repeat the process. Remember, statement 1 does the following: (1) causes a group of statements to be repeatedly executed and (2) determines when the repetition has come to an end. To carry out this second task, control must go back to statement 1 after statement 5.

Every time control returns from statement 5 to statement 1, a fundamental inquiry is raised by the computer: has the group known as *read-and-print-process* been executed ten times? If the result of that inquiry is no, then the computer performs that group of statements again. If the answer is yes, on the other hand, then the computer transfers control to statement 2 instead. Statement 2 terminates the execution of the program. Again, iterative structures, such as statement 1, cause a select group of statements to be repeatedly executed a stated number of times.

Fixed Termination of the Iteration

The iterative structure shown in figure 2.7 has a very distinct limitation. A program using this particular version of the iterative structure can read and print only a total of ten record images. This requires the program to be modified if fewer, or more, record images are to be read. The alteration is minor—only the number 10 needs to be changed. However, a program that must be modified every time the number of input data record images changes leaves much to be desired. What would happen if you incorrectly count the number of record images to be processed? In such circumstances, your program would attempt to process either fewer or more record images than you have actually supplied. An attempt to process more records would cause an abnormal termination of your program. An attempt to print fewer records would cause the program to simply not print all of the record images you wanted. Either way, the results are unsatisfactory.

One way to eliminate the need for counting the input record images to be processed is to establish the practice that the last record to appear in the input file be unique. This unique record serves as the file terminator. For now, such a record will contain only blanks.

Figure 2.7
Limitation of iterative structure

```
do the read and print process
        10 times.
```

Figure 2.8
Pseudo-code with revi
iterative structure

```
Statement        Pseudo-coded statement
 number

   1             do the read and print process
                        until
                              record read is all blank
   2             stop execution.

   3        read and print process
   4             read another inventory record image
   5             print that very same record image.
```

Variable Termination of the Iteration

By using a file terminator record, our program can detect when the end of the file has been reached without fear of reading the wrong number of record images. Of course we have to exercise great care that this terminator record always appears last within the input file. Remember, the program is being restructured to end its execution whenever the terminator record is read. If the terminator record is the first record in the input data file, then none of the remaining record images will be read.

The revised program logic could be altered to look like that shown in figure 2.8. Note the different form of the iterative structure in figure 2.8, shown in statement 1. In this version, the read-and-print-process continually executes until a blank record is encountered. A program designed in this manner can successfully process every record in the input file no matter the size of the file. All that is required is that the last record in the file be absolutely blank.

The program's logic contains one last problem in its most recent form. Specifically, the last record read is the blank record. As the read-and-print-process is now structured, that last record will be read (statement 4) and then printed (statement 5). Do we really want to print the blank record? You might reasonably respond that when this blank record is printed it results in a blank line at the end of the report and would not be noticed.

An extra blank line at the very end of a report is not a terrible problem. However, the logic of the program fails to recognize that the last record, our terminator, is not to be processed in the same manner as were all of the preceding record images. To overcome this flaw in construction, the alternative structure in figure 2.9 is a possible solution. The revised program logic shown in figure 2.9 differs markedly from our initial attempt in this area. Note that the read-and-print-process has been renamed and reconstituted to become the *print-and-read-process.* Now printing occurs prior to reading the record image, or so it might seem.

Remember, the restructuring effort is required because the terminator record is being processed in the same fashion as all of the other record images. This shortcoming has now been remedied by making the read instruction the last instruction in the group of statements named *print-and-read-process.* In general, no record image read in this group is printed immediately after it has been read. Instead, control returns to the iterative structure in statement 2. If the record image just read is blank, then the iterative structure relinquishes control of the computer and statement 3 assumes control for the purpose of terminating the program's execution. If the record image is not blank, then the iterative structure returns control to the group of instructions named *print-and-read-process.* The first instruction in that group is to print the nonblank record image read the last time control was in this group of statements.

Statement number		Pseudo-coded statement
1		read very first inventory record image.
2		do the print and read process until record read is blank.
3		stop execution.
4	print	and read process
5		print that very same record image.
6		read another inventory record image

Figure 2.9
Revised reading/printing process

The only problem with this revised system of statements is that the print-and-read-process expects a record image to have already been read when this group first receives control of the computer. This is satisfactory the second, third, and so on times the iterative structure transfers control to the group. It is not satisfactory, however, the very first time control reaches the group. As a result, the process is primed by initiating the very first read operation prior to the iterative structure. Doing so guarantees that the process will have a record image to print when it receives control the first time it is executed.

Summary

All report programs are based on the same skeletal structure. While the titles produced by a report program are unique to that report, the manner in which records are read and printed is always the same. This common structure permits us to describe the basic inner workings of a report program using an English-like language known as pseudo-coding. By verifying the correctness of our program's logic, using a simple set of statements written in a familiar language, we can identify errors well before we start the use of COBOL. Errors found and corrected early in program design reduce the amount of effort required to complete the programming assignment.

Report programs are iterative in nature. They involve the printing of record images from an input file. It doesn't matter whether the input file contains inventory records, payroll records, or whatever. The repetitive workings of the program remain the same.

At the heart of every report program is an iterative structure. This particular instruction causes program execution to continue until a specified termination condition is met. In our example thus far, the requirement for terminating the program's execution is the reading of the terminating record in the input file. By having a terminating record, our program is flexible enough to print any and all record images that we may choose to process. The only requirement is that the terminating record be the last record in the input file. We may, on different occasions, have our program process 212 record images, followed by 654 images the very next time. The program, however, need not be changed to reflect the differing sizes of the input file. The terminating record image provides this capability.

The terminating record image should not be processed in the same manner as the record images preceding it in the input file. Unlike real record images, the terminating record should not be printed on the report. For that reason, the program logic for all report programs should, in its most recent state, be similar to the one illustrated in figure 2.10.

Figure 2.10
Final version of report program

```
Statement        Pseudo-coded statement
 number

   1                 read very first inventory record image.
   2                 do the print and read process
                          until
                               record image is blank.
   3                 stop execution.

   4           print and read process
   5                 print that very same record image.
   6                 read another inventory record image
```

Notes

1. Actually, the number of printed lines would be somewhat larger. This estimate fails to account for the printed lines that contain report titles, etc.

Questions for Review

1. Why do all report programs share the same basic structure?
2. Normally, control of the computer is directed from one instruction to the very next. That is, control goes from statement 1 to 2, to 3, and so on. Is this true of all statements?
3. Why is the logic of the report program modified to permit the use of a flexible termination rather than a fixed termination?
4. Why was the term *read-and-print process* changed to *print-and-read process?*
5. Why must a record from the input file be read before the print-and-read process is invoked for the first time by the iterative statement?

Problems

1. Examine the following pseudo-coded version of a report program. What, if anything, is wrong?

```
Statement       Pseudo-coded Statement
 Number

   1                  read very first inventory record image.
   2                  do the read and print process
                             until
                                  record image is blank.
   3                  stop execution.
   4            print and read process
   5                  read another inventory record image
   6                  print that very same record image.
```

2. Examine the following pseudo-coded version of a report program. What, if anything, is wrong?

```
Statement       Pseudo-coded Statement
 Number

   1                  do the read and print process
                             until
                                  record image is blank.
   2                  stop execution.
   3            print and read process
   4                  print that very same record image.
   5                  read another inventory record image.
```

3. Examine the following pseudo-coded version of a report program. What, if anything, is wrong?

```
Statement       Pseudo-coded Statement
 Number

   1                  read very first inventory record image.
   2                  do the read and print process
                             until
                                  record image is blank.
   3            print and read process
   4                  read another inventory record image
   5                  print that very same record image
```

3 Translation of the Pseudo-Code to COBOL

Chapter Outline

In chapter 2 the pseudo-coded version of the basic program logic for all report programs was developed. The effect of each of the program's statements was discussed. For instance, it was noted that iterative instructions cause the normal sequential flow of control from instruction to instruction to be disrupted. The iterative structure takes control out of the normal instruction sequence into a group of instructions not physically adjacent to the iterative instruction.

All of those discussions may lead you to believe that the computer can execute pseudo-code directly. This is not the case at all. Pseudo-code is a convenient tool for developing program logic without the strict syntax rules of the COBOL language. Sooner or later, however, you must convert your pseudo-coding into a programming language that can actually be operated on by the computer. The time has come to convert the pseudo-coded report program into COBOL.

Basic COBOL Program Construction

Every COBOL program is made up of four divisions. These divisions are called (1) the IDENTIFICATION DIVISION, (2) the ENVIRONMENT DIVISION, (3) the DATA DIVISION, and (4) the PROCEDURE DIVISION. Each of these divisions has a specific function, and each must be present in every COBOL program.

COBOL programs consist of many horizontal lines or record images. Each line contains, more or less, a single COBOL definition or instruction. These source entries are added to the program a line at a time by the computer programmer. The record images that constitute a COBOL program are each 80 positions in length. Some programming languages feature great flexibility in permitting the programmer to decide where each definition or instruction is to be started. PL/I, for example, permits definitions and instructions to be started in column 1, 40, or even 80. COBOL, however, is somewhat less permissive.

Each statement in a COBOL program is divided into four separate segments. We will cover two of these segments at this point. The *A-Margin* is located in columns 8 through 11 of each COBOL source statement. The *B-Margin* is located in columns 12 through 72. The A-Margin is used to hold the names of the divisions just discussed and also section and paragraph names. The B-Margin is designed to hold the actual COBOL verbs that direct the actions of the computer. Note the pseudo-coded version of all report programs reproduced in figure 3.1.

A-Margin
Positions 8 through 11 of each line, or source record, in a COBOL program. Used to contain division, section, and paragraph entries.

B-Margin
Positions 12 through 72 of each line, or source record, in a COBOL program. Used to contain COBOL instructions that direct computer actions.

Figure 3.1
Report program logic

```
Statement        Pseudo-coded statement
 number

   1             read very first inventory record image.
   2             do the print and read process
                       until
                             record read is blank.
   3             stop execution.

   4       print and read process
   5             print that very same record image.
   6             read another inventory record image
```

Figure 3.2
PROCEDURE DIVISION
equivalent of pseudo-code

```
       PROCEDURE DIVISION.
       MAIN-CONTROL SECTION.
       M-C-001.
           ACCEPT RECORD-IMAGE.
           PERFORM PRINT-AND-READ-PROCESS
               UNTIL
                   RECORD-IMAGE = SPACES.
           STOP RUN.
       PRINT-AND-READ-PROCESS SECTION.
       P-A-R-P-001.
           DISPLAY RECORD-IMAGE.
           ACCEPT RECORD-IMAGE.
```

Procedure Division

PROCEDURE DIVISION
The division composed of sections that consist of paragraphs that, in turn, contain executable instructions. The fourth of four divisions in every COBOL program.

The most convenient place to start the conversion of pseudo-code to COBOL is with the PROCEDURE DIVISION. The COBOL equivalent of the pseudo-code shown in figure 3.1 is illustrated in figure 3.2. As figure 3.2 illustrates, certain items begin in the A-Margin while others can only begin in the B-Margin. (The statement formats, which appear in this chapter and throughout the text, show the skeletal structure of a statement. The notations indicate the margin or column in which the statement begins.)

Statement Format

```
A
↓
PROCEDURE DIVISION.
```

Sections and Paragraphs
The PROCEDURE DIVISION is divided into sections. Sections, in turn, are composed of paragraphs. Figure 3.2 shows two such sections, each of which corresponds to a major portion of our program. The MAIN-CONTROL SECTION is responsible for reading the very first input record, iteratively processing the remaining records, and stopping the execution of the program when all the input records have been processed. At the moment, this section is made up of only one paragraph. That paragraph is named M-C-001. As we further improve our report program, we will add several more paragraphs to this particular section. The PRINT-AND-READ-PROCESS SECTION is structured along similar lines.

Statement Format

```
A
↓
sectionname SECTION.
```

Sections
A means by which related COBOL instructions can be identified according to program function.

Paragraphs
A means by which sections can be partitioned into smaller, easier to understand units of COBOL programs.

Sections identify major functions of a COBOL program. *Paragraphs* identify major subfunctions within sections. Since a section may be made up of many subfunctions, it is likely to contain more than one paragraph, although this need not be true in all cases.

Statement Format

```
A
↓
paragraphname.
```

A convention used throughout this book is that paragraph names are abbreviations of the sections in which they are found. Because a section can contain many paragraphs, each paragraph name is suffixed with a numeric value. For instance, M-C-001 indicates that this is the first (001) paragraph in the section MAIN CONTROL.

Section and paragraph names are governed by similar rules. First of all, these names may not exceed thirty characters in length. Very explicit section and paragraph names can be constructed within this limitation of thirty characters. Second, these names may contain only three types of characters: numerals (0–9), letters of the alphabet (A–Z), and the hyphen.[1] These characters can be combined in any order with the following exceptions.

1. A name may not begin with a hyphen. Names can begin only with numerals or letters of the alphabet.
2. A name may not end with a hyphen. Names must have either numerals or letters of the alphabet as their last character.
3. A name may not contain two or more hyphens in a row. Names may contain as many hyphens as you would like, but these hyphens cannot appear one after the other.

Section names must be followed by at least one blank space and the word SECTION. Also, all section and paragraph names are terminated by a period.

Only one additional warning is needed concerning section and paragraph names. The designers of COBOL have made it impossible for COBOL programmers to choose certain names for sections and paragraphs. These names are said to be *reserved.* A list of reserved names can be found in the technical literature that describes the COBOL compiler used at your installation. When you are constructing a COBOL program, be sure to consult this list often. You cannot use a name for a section or paragraph if it can be found within this list. It should be noted, however, that even reserved names can be transformed into legal names by slight alteration. Possible methods of alteration include, but are not limited to, slight changes in spelling, insertion of hyphens, removal of hyphens, and numeric suffixes.

Reserved Words
COBOL verbs, such as ACCEPT, that cannot also be used in any other context. Because of their special purposes, reserved words cannot be used as section, paragraph, or variable/field/record names.

Terminating Execution: STOP RUN

Perhaps the simplest COBOL instruction is the last one to be executed in any COBOL program. This instruction, when executed, causes the program to terminate. Once terminated, that program automatically relinquishes the machine resources it has been allocated.

A review of the pseudo-coded version of our report program in figure 3.3 shows the instruction. This pseudo-coded instruction is replaced with the COBOL instruction STOP RUN. Note that this COBOL verb starts in the B-Margin of the COBOL program source statement. Furthermore, the verb is terminated by a period.

Statement Format

```
B
↓
STOP RUN.
```

Statement number	Pseudo-coded statement
.	.
.	.
3	stop execution.
.	.

Figure 3.3
Stop instruction in pseudo-code

Figure 3.4
Input operations in the program logic

```
Statement        Pseudo-coded statement
 number

    1                 read very first inventory record image.
    .                 .
    .                 .

    4          print and read process
    .                 .
    6                 read another inventory record image
```

Input Operations: ACCEPT

The pseudo-coded version of our report program in figure 3.4 requires the use of two input operations. The first input operation is responsible for reading the very first record required by the PRINT-AND-READ-PROCESS. The second read is located within this group of instructions. Records which are read through this particular input operation are printed the next time the group of instructions is executed by the iterative structure. Both read instructions are replaced by the same COBOL verb. The form of that verb is ACCEPT RECORD-IMAGE. As with STOP RUN, the name of the verb is the first part of the instruction. Also, the instruction is terminated by a period and must be wholly contained in the B-margin. Immediately following the verb is the name of the computer memory area in which the contents of an 80 column record image are to be held. By giving that area of computer memory a name, the contents of the record read can be referred to symbolically, in this case, as RECORD-IMAGE. RECORD-IMAGE is also known as an identifier, *variable,* or record.

Variable
A symbolically named collection of primary memory locations into which data may be stored and subsequently retrieved. (It should be noted that records consist of variables.)

Statement Format

```
B
↓
ACCEPT variablename [FROM mnemonic].
```

Variable names, like section and paragraph names, are chosen by the COBOL programmer. The rules to be followed are exactly the same as those used for creating section and paragraph names. As was mentioned earlier, be sure not to choose a name for a variable that can be found in the list of reserved names.

This particular instruction causes the destruction of whatever the contents of RECORD-IMAGE happened to be prior to its execution. In other words, every time a new record image is read, it is placed in the same area of computer memory as the last record image was placed. The area of memory is only big enough to hold one record image at any one time, so the most recent record read is the only record image retained. All earlier record images are destroyed with each new input operation.

Output Operations: DISPLAY

Unlike the input operation, our report program needs only one output instruction, as shown in figure 3.5. Both input operations cause the image of an input record to be placed into an area of computer memory symbolically referred to as RECORD-IMAGE. In order to print the contents of RECORD-IMAGE, the COBOL verb DISPLAY RECORD-IMAGE is required. Again, the verb name DISPLAY appears first, the variable name second, and the period last. As with all verbs, this must be wholly contained within the B-Margin.

Statement Format

```
B
↓
DISPLAY variablename [UPON mnemonic].
```

Statement number	Pseudo-coded statement
.	.
.	.
.	.
4	print and read process
5	print that very same record image.
.	.

Figure 3.5
Output operations in program logic

Statement number	Pseudo-coded statement
.	.
2	do the read and print process until record read is blank.
.	.
.	.

Figure 3.6
Iterative instruction in pseudo-code

```
           PERFORM PRINT-AND-READ-PROCESS
                   UNTIL
                         RECORD-IMAGE = SPACES.
```

Figure 3.7
COBOL PERFORM verb

The computer causes the printer to advance one line and then print the contents of the variable named in the DISPLAY instruction. Unlike the ACCEPT instruction, the DISPLAY instruction does not cause the contents of RECORD-IMAGE to be altered. As a result, the record image placed into RECORD-IMAGE by the ACCEPT instruction remains after the DISPLAY instruction has completed its function.

Iterative Operation: PERFORM

At the heart of the pseudo-coded version of the report program is the iterative instruction that causes the printing of every nonblank record in the input file. This particular pseudo-coded instruction, in figure 3.6, is replaced by the COBOL instruction shown in figure 3.7. The name of the verb PERFORM comes first. Following the verb is the name of the COBOL section containing the paragraphs and instructions to be executed. In this case, the section to be executed is PRINT-AND-READ-PROCESS. This particular section contains only one paragraph P-A-R-P-001. This paragraph, in turn, contains only two instructions, DISPLAY and ACCEPT. Consequently, this PERFORM verb causes all of the verbs (DISPLAY and ACCEPT) in all of the paragraphs (P-A-R-P-001) that are part of the section (PRINT-AND-READ-PROCESS) to be executed.

The name of the section containing the instructions to be executed is followed by the condition to end the iteration. The COBOL language requires that the word UNTIL preface the required condition. The actual termination specification is RECORD-IMAGE = SPACES. RECORD-IMAGE is, you should remember, the name of the computer memory area in which all record images are held after completion of the ACCEPT instruction. You should also recall that whenever a blank record image is encountered, it is assumed that the last record in the input file has just been read. Since the file is exhausted, the end of the iterative process occurs.

Statement Format

```
B
↓
PERFORM sectionname [UNTIL condition].
```

This particular verb, PERFORM, unlike the earlier verbs, has been written on three separate lines, or source statements.

As chapter 4 notes, the format of any COBOL program is extremely important. Programmers should exercise great care in formatting the COBOL statements in their programs to aid interpretation by the reader. Note the less expansive alternative shown in figure 3.8. This particular version requires fewer source statements. In fact, the whole instruction could be written on a single source statement. While fewer statements are preferred by the key operator, they are not likely to be as well received by the reader. Use enough statements to make your instructions easy to read.

Any COBOL instruction can be spread out over more than one source statement should you choose to do so. Figure 3.9 provides such an example. While somewhat absurd, the example in figure 3.9 illustrates two very important points. First, the programmer has great choice in program format. Second, a carelessly conceived format can definitely impair the reader's ability to comprehend the program. To further illustrate this point, note one last example. The illustration in figure 3.10 shows that more than one COBOL verb can be placed on a single source statement. While much effort might be saved in the initial preparation of a program written in this way, these savings are only at great cost to those who must read and comprehend the logic of this program.

Identification Division

All COBOL programs are composed of four divisions. Thus far you have seen only one, the PROCEDURE DIVISION, so you have seen incomplete COBOL programs.

IDENTIFICATION DIVISION
The first of four divisions in every COBOL program. It serves to identify the circumstances that lead to the program's creation and use.

The IDENTIFICATION DIVISION is the first division to appear in a COBOL program. Its use is required and can be seen in figure 3.11. This division statement also begins in the A-Margin as does the PROCEDURE DIVISION.

Statement Format

```
A
↓
IDENTIFICATION DIVISION.
```

The purpose of this particular COBOL division is to identify many of the facets of a COBOL program. The only part of this division we will explain in this chapter is the need to give the program a name. This name must also start in the A-Margin, and is a statement with the PROGRAM-ID. RPT0301. An examination of this COBOL statement shows the use of two periods. The first follows the paragraph PROGRAM-ID. The second follows the name of the program. The program's name can be no longer than eight characters in length. Otherwise, the rules for forming program names are identical to the rules for section, paragraph, and variable names. At least one blank space should separate the first period from the first character of the program's name. Again, reserved words must be avoided.

```
    PERFORM PRINT-AND-READ-PROCESS UNTIL RECORD-IMAGE
    = SPACES.
```

Figure 3.8
Poorly formatted PERFORM verb

```
    .
    .
    ACCEPT
    RECORD-IMAGE.
    PERFORM
    PRINT-AND-READ-PROCESS
    UNTIL
    RECORD-IMAGE
    =
    SPACES.
    .
    .
```

Figure 3.9
Spreading COBOL instructions over many source statements

```
    .
    .
    ACCEPT RECORD-IMAGE. PERFORM PRINT-AND-READ-PROCESS
    UNTIL RECORD-IMAGE = SPACES.
    STOP RUN.
    .
    .
```

Figure 3.10
Squeezing COBOL instructions onto few source statements

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    RPT0301.
 .
 .
PROCEDURE DIVISION.
MAIN-CONTROL SECTION.
M-C-001.
    ACCEPT RECORD-IMAGE.
    PERFORM PRINT-AND-READ-PROCESS
        UNTIL
            RECORD-IMAGE = SPACES.
    STOP RUN.
PRINT-AND-READ-PROCESS SECTION.
P-A-R-P-001.
    DISPLAY RECORD-IMAGE.
    ACCEPT RECORD-IMAGE.
```

Figure 3.11
Addition of the IDENTIFICATION DIVISION

Figure 3.12
IDENTIFICATION, ENVIRONMENT, and PROCEDURE DIVISIONS

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.    RPT0301.
       ENVIRONMENT DIVISION.
         .
         .
         .
       PROCEDURE DIVISION.
       MAIN-CONTROL SECTION.
       M-C-001.
           ACCEPT RECORD-IMAGE.
           PERFORM PRINT-AND-READ-PROCESS
               UNTIL
                   RECORD-IMAGE = SPACES.
           STOP RUN.
       PRINT-AND-READ-PROCESS SECTION.
       P-A-R-P-001.
           DISPLAY RECORD-IMAGE.
           ACCEPT RECORD-IMAGE.
```

Environment Division

ENVIRONMENT DIVISION
The second of four divisions to be found in every COBOL program.

For people who have trouble spelling, this particular COBOL division may prove to be a problem. While the ENVIRONMENT DIVISION must be supplied in all the COBOL programs we will illustrate, the statements that comprise it will not be discussed until the chapters illustrating conventional file structures. In the meantime, our use of the ACCEPT and DISPLAY verbs takes the place of the statements normally used in this division. Our COBOL program now appears as shown in figure 3.12.

Statement Format

```
A
↓
ENVIRONMENT DIVISION.
```

Data Division

DATA DIVISION
The third of four divisions in every COBOL program. All field, variable, and record definitions are located within this part of the COBOL program.

The DATA DIVISION is the division in which all the variables used in a COBOL program must be defined. Like the PROCEDURE DIVISION, this particular division is composed of sections. The only section we will discuss, however, is the WORKING-STORAGE SECTION.

Statement Format

```
A
↓
DATA DIVISION.
```

Like all sections names thus far, this section name must begin in the A-Margin. An attempt to define any variable in COBOL programs without this section name will end in disaster. This section must appear in all COBOL programs.

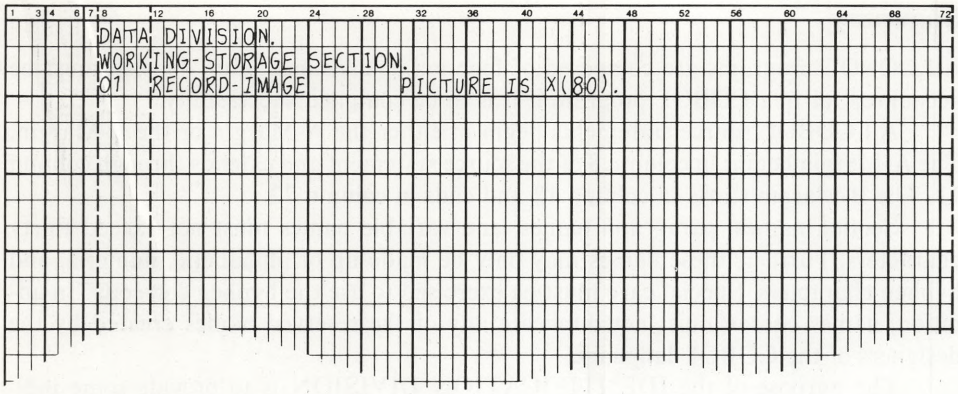


Figure 3.13
Adding the DATA DIVISION

Like the word ENVIRONMENT, programmers frequently misspell WORKING-STORAGE. It is very easy to omit the hyphen that separates the two components of the word WORKING-STORAGE. And, without that hyphen, the section name is unrecognizable to the computer.

Statement Format

```
A
↓
WORKING-STORAGE SECTION.
```

The format of this division, its section, and a sample variable declaration are shown in figure 3.13. Since RECORD-IMAGE is a variable used in the PROCEDURE DIVISION, it must be defined in the WORKING-STORAGE SECTION of the DATA DIVISION.

All variable definitions begin with the numerals 01 in the A-Margin. The notation 01 indicates that the variable being defined is not described field-by-field. You should recall that the inventory and time card records were composed of many different fields. COBOL permits a programmer to describe a record field-by-field. For now, however, records will not be defined field-by-field. Instead, we will define the record as one large field, and in this case, a field that is 80 characters long.

The name of the variable must begin in the B-Margin. The last character of the name must be followed by at least one blank space. The phrase PICTURE IS must follow this blank space. What does PICTURE IS imply?

Statement Format

```
A   B
↓   ↓
01  variablename PICTURE IS attributes.
```

Every variable in a COBOL program is said to possess attributes. These attributes describe how much computer memory has been allocated to the variable, or how big it is. Attributes also describe the type of data the variable can hold. Consequently, X(80) tells us that the variable named RECORD-IMAGE can hold up to 80 characters worth of information. Coincidentally, this is the size of a punched card. Further, the letter *X* denotes that each of those 80 positions can be any letter of the alphabet, numeral, space, or special character. As is always the case, the definition must be terminated by a period. In addition, the letter *S* in IS must be separated by at least one blank space from the *X* in X(80). Also, no blanks can be embedded in the attribute list X(80).

Summary

All COBOL programs are composed of four major divisions. Each division must be accounted for in a COBOL program if it is to be compiled successfully.

All COBOL source statements begin in either the A-Margin or the B-Margin. The A-Margin begins in column 8 of the source statement and extends through column 11. The B-Margin starts in column 12 and ends in column 72.

Section names, paragraph names, and variable names must not exceed thirty characters. Further, these names may only be made up of alphabetic, numeric, and hyphen characters. Special care must be exercised to use the hyphen correctly. In addition, names chosen must not duplicate any of the reserved names created by the designers of the COBOL language.

The purpose of the IDENTIFICATION DIVISION is to provide some indication of the program's name. This name must adhere to the naming conventions just summarized except that the maximum length is 8 characters.

While not fully explained at this time, the ENVIRONMENT DIVISION must appear in all COBOL programs. Its sections will be discussed in subsequent chapters relating to machine environments and conventional file instructions.

The DATA DIVISION is that division in which all variables must be defined. These definitions list the attributes of the variable. Attributes include the specification of the record's length and the type of data it can hold. At present, alphanumeric fields are the only fields we know how to declare.

Finally, the PROCEDURE DIVISION is comprised of sections, paragraphs, and instructions. Each section corresponds to a major function within the COBOL program. Each function may, in turn, be represented by more than one paragraph. A paragraph corresponds to a subfunction.

The formatting of the instructions in the PROCEDURE DIVISION is critical to the reader of that COBOL program. Programmers are advised to place COBOL instructions in such a manner as to maximize comprehension by the reader.

The final form of our COBOL program is shown in figure 3.14.

Figure 3.14
Final COBOL report program

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.  RPT0302.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  RECORD-IMAGE    PICTURE IS X(80).
       PROCEDURE DIVISION.
       MAIN-CONTROL SECTION.
       M-C-001.
           ACCEPT RECORD-IMAGE.
           PERFORM PRINT-AND-READ-PROCESS
                UNTIL
                      RECORD-IMAGE = SPACES.
           STOP RUN.
       PRINT-AND-READ-PROCESS SECTION.
       P-A-R-P-001.
           DISPLAY RECORD-IMAGE.
           ACCEPT RECORD-IMAGE.
```

Notes

1. Other characters may be possible. Whether they are or not is a feature of the particular COBOL compiler being used. Check your installation literature that describes the COBOL compiler available and the characters that may be used.

Questions for Review

1. How many divisions must a COBOL program contain? Name each division and its purpose.
2. What, if anything, is wrong with the following possible section, paragraph, variable, and program names?

```
0001-MAIN-CONTROL     TRANSACTION--FIELD
RECORD-READ           -RECORD-IMAGE-
CARDIMAGE             THIS-IS-THE-RECORD-READ-NOW-001
CARD IMAGE            WORKING STORAGE
```

3. What is the advantage in formatting a COBOL instruction, like PERFORM, so that it requires three lines in a COBOL program rather than just one?
4. Must the divisions of a COBOL program appear in any particular sequence? If so, what?
5. What advantage is gained if paragraph names are abbreviated versions of the sections in which they belong?

Problems

1. Examine the following COBOL program very carefully. Locate as many errors as you possibly can. Suggest a corrected version.

```
IDENTIFICATION DIVISION.
ENVIORNMENT DIVISION
DATA DIVISION.
WORKING STORAGE SECTION
    01  RECORD-IMAGE X (80).
PROCEDURE DIVISION.
MAIN-CONTROL-SECTION.
M-C-001
   ACCEPT.
    PERFORM READ-AND-PRINT PROCESS
        UNTIL
            RECORD-IMAGE SPACES.
    STOP EXECUTION.
PRINT-AND-READ-PROCESS SECTION.
P-A-R-P-001.
    DISPLAY CARD IMAGE
    ACCEPT CARD IMAGE.
```

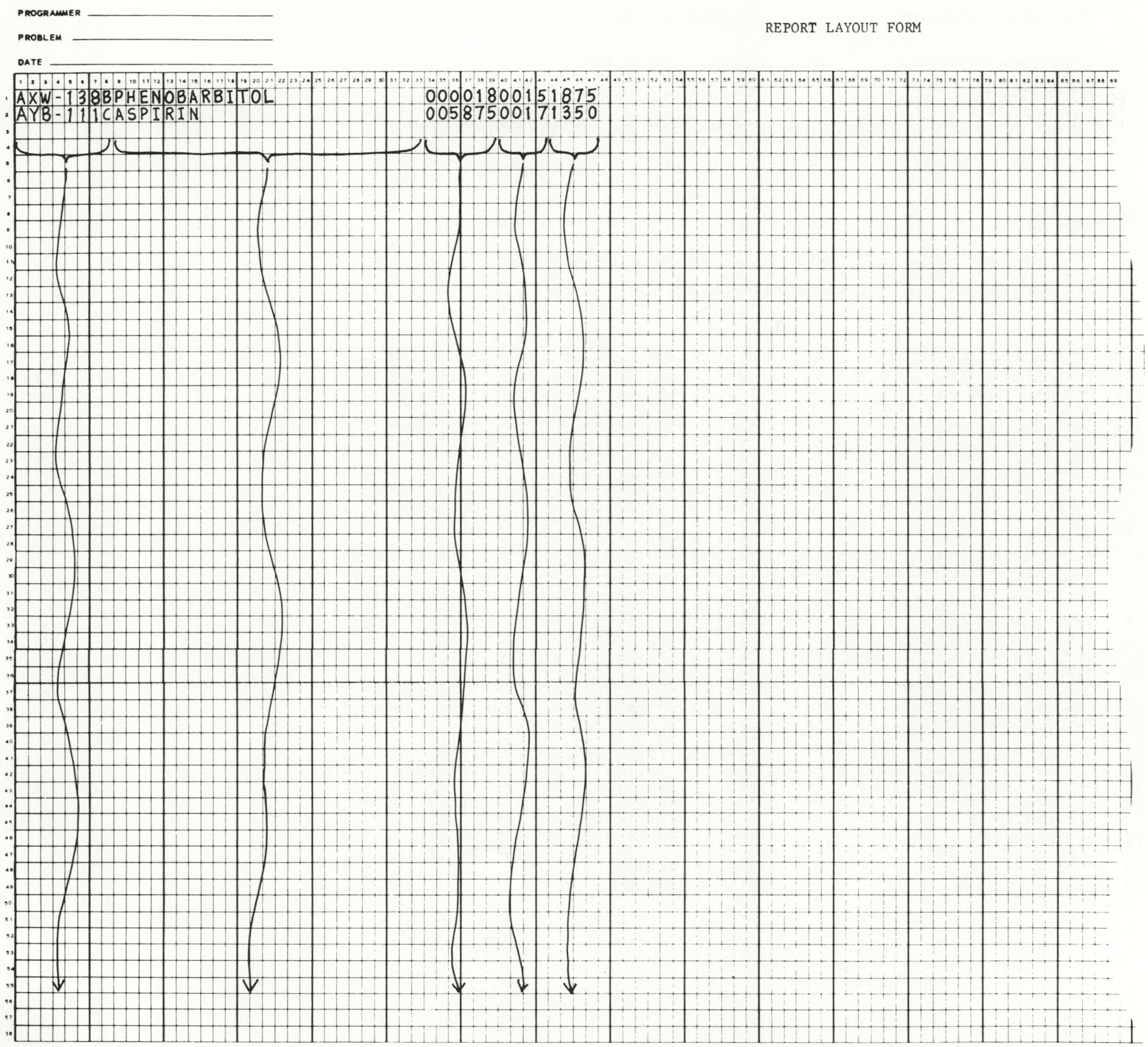
Programming Assignments

1. Wexler Drug Distributors requires a report program to produce a report that lists the records that constitute its Drug Master File. Each record in the file can be described as follows:

Record Position	*Data Type*	*Variable*
01–08	N	Drug identification code
09–33	AN	Drug name
34–39	N	Units on hand
40–43	N	Acquisition cost/unit (cents)
44–47	N	Resale cost/unit (cents)
48–80	—	Unused

The report produced should appear similar to the one shown in figure 3.15.

Figure 3.15
Wexler Drug Distributors drug report

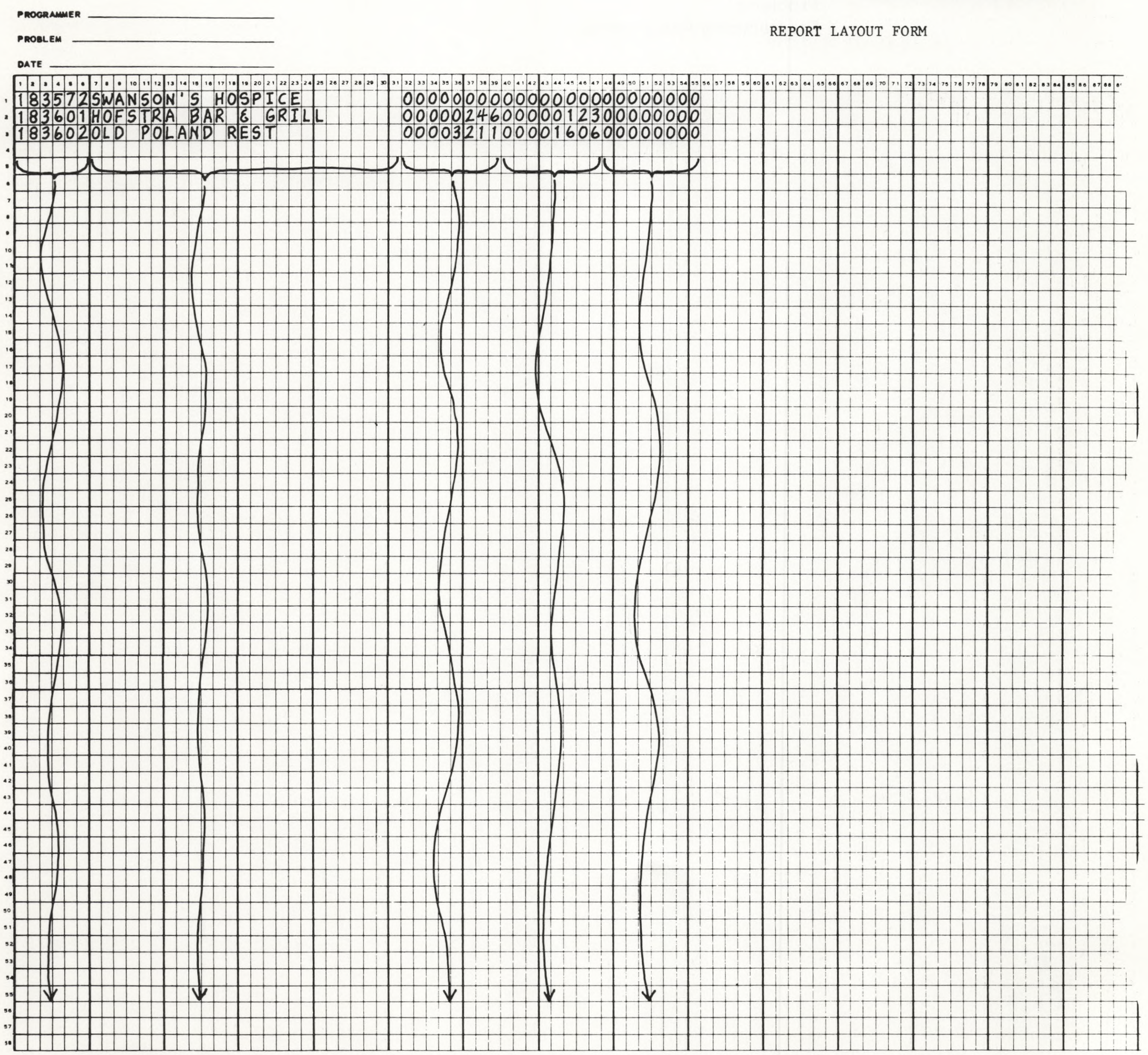


2. Henderson Liquor Supply requires a report that lists the records that constitute its Customer Credit File. Each record in the file can be described as follows:

Record Position	*Data Type*	*Variable*
01–06	N	Customer credit identification number
07–31	AN	Customer's name
32–39	N	Current balance ($)
40–47	N	Monthly payment due ($)
48–55	N	Past due amount ($)
56–80	—	Unused

The report produced should appear similar to that shown in figure 3.16.

Figure 3.16
Henderson Liquor Supply customer credit report



4 Comments as Documentation

Chapter Outline

The report program structured in chapter 3 is not truly representative of report programs constructed in the data processing field. It is a very simplified 80–80 listing program introducing you to the process of building a COBOL computer program from a basic model. Most real COBOL computer programs are much more complex than the 80–80 listing program. A program, whose structure and function is more complex, requires additional documentation for the reader of that program.

Programming documentation takes many forms, such as pseudo-coded descriptions, flowcharts, facsimile reports, record layouts, and most importantly, the source program listing produced by the COBOL compiler. The source program listing is perhaps the device most frequently used to understand a program's inner workings. Unless the program logic illustrated in a program listing can be readily understood by the reader, efforts to maintain or enhance that program will be very difficult. One method of improving the reader's comprehension is to include high-quality descriptive narrative in the form of *comments.*

Comments
The inclusion of English narrative as part of the COBOL source program to improve reader's comprehension of program functions.

The Need for Commenting

At one time COBOL was considered a self-documenting language. That is, the English-like nature of its statements led a number of people within the data processing field to assert that the source listing produced by a COBOL compiler was sufficient documentation for the reader to understand that program's functions. This particular viewpoint has changed. The data processing industry has since learned that it is the way in which these executable statements are combined that gives any computer program its structure, or logical flow. Consequently, while each instruction may be English-like, the combinations of these individual, easily-read statements are likely to be incomprehensible. The number of different ways these executable statements may be combined to produce a working COBOL program appears to be infinite. Since so many versions of the same computer program are possible, an additional aid must be included to assist the program interpretation process. The best additional aid that can be supplied by any programmer is narrative commentary.

Computer programs can be structured, or partitioned, into logical components. Each component performs a specific function or set of functions. These functions, however, may not be obvious to the reader. For that reason, each logical segment of a COBOL program must be prefaced by a series of descriptive comments telling the location, purpose, and peculiarities of each segment. With these comments, the programmer provides the reader with an invaluable tool.

When each logical segment of the program is broken out of the program by the use of comments, the overall structure of that program can be understood by the reader. Further, a series of comments can accurately describe (1) the relationship of that segment to the remainder of the program, and (2) the specifics of that module, such as how it works. The reader can then better understand the logic adopted by the programmer.

You might be wondering who would be interested in reading your programs. The question is a valid one. The answer is, the person who will maintain your programs.

The greatest expense in initially developing computer software is not the computer time involved in creating the program. Electronic technology has reduced computer costs significantly. The greatest expense is the salary of the programmer assigned to develop the program. Your salary as a programmer costs the organization more than the computer resources you happen to use.

The expense of creating a computer program, however, is still not its greatest cost component. Maintaining that program costs more than its initial development. What is maintenance?

Figure 4.1
COBOL report program without comments

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. RPT0401.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  RECORD-IMAGE          PICTURE IS X(80).
       PROCEDURE DIVISION.
       MAIN CONTROL SECTION.
       M-C-001.
           ACCEPT RECORD-IMAGE.
           PERFORM PRINT-AND-READ-PROCESS
               UNTIL
                   RECORD-IMAGE = SPACES.
           STOP RUN.
       PRINT-AND-READ-PROCESS SECTION.
       P-A-R-P-001.
           DISPLAY RECORD-IMAGE.
           ACCEPT RECORD-IMAGE.
```

As you may have gathered in your own programming experience, programs do not always work properly every time they are executed. In fact, a corollary to Murphy's Law puts this last statement in its proper perspective. Specifically, every program has a last "bug" in it, and that bug has not yet occurred. No matter how skillful you are in designing and implementing a computer program, you will never create a fail-safe program. Therefore, you should always document your programs with comments as you are writing them so that in the event of their failure, you or anyone else can understand the logic with which they were written. Even the 80–80 listing program discussed in chapter 3, which is a very simple program, should have included comments to explain the logic. No program, whatever its degree of complexity can be considered complete by the programmer until each of its logical segments has been appropriately commented. A review of that program illustrates that it contains two primary logical segments.

The first segment, shown in figure 4.1, is said to represent the program's overall control function. Here the PERFORM instruction guides the execution of the PRINT-AND-READ-PROCESS SECTION. Record images are read and subsequently printed until no more record images remain to be processed. The second logical segment is the PRINT-AND-READ-PROCESS SECTION itself. The executable statements within this paragraph actually carry out the printing and reading operations. Proper commenting of this particular computer program involves restating, in a simplified manner, the descriptions of these logical segments. In addition, an explanation of the program's function should be stated somewhere in the program. For now, we shall place that explanation of purpose at the very beginning of the PROCEDURE DIVISION. The COBOL program in figure 4.2 illustrates the modified appearance of the 80–80 listing program with these suggested comments. The program shown in figure 4.2 is, hopefully, easier to read than its earlier, uncommented counterpart.

Page 1 of 2

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. RPT0402.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  RECORD-IMAGE          PICTURE IS X(80).
       PROCEDURE DIVISION.
       MAIN-CONTROL SECTION.
       M-C-001.
           NOTE    THIS SECTION OF THE PROGRAM SERVES TO
                   CONTROL THE PRINT AND READ PUMP SECTION
                   THAT FOLLOWS; TO START THE PUMP ONE
                   RECORD IMAGE IS READ AND THEN THE PUMP
                   IS EXECUTED; WHEN THE BLANK RECORD IS
                   FOUND, THE PUMP TERMINATES ALONG WITH
                   PROGRAM EXECUTION.
           ACCEPT RECORD-IMAGE.
           PERFORM PRINT-AND-READ-PROCESS
               UNTIL
                   RECORD-IMAGE = SPACES.
           STOP RUN.
       PRINT-AND-READ-PROCESS SECTION.
       P-A-R-P-001.
           NOTE    THIS SECTION REPRESENTS THE PRINT AND READ
```

Page 2 of 2

```
                   PUMP MENTIONED EARLIER; WHEN INVOKED IT
                   PRINTS THE PREVIOUSLY READ RECORD IMAGE
                   AND THEN READS YET ANOTHER.
           DISPLAY RECORD-IMAGE.
           ACCEPT RECORD-IMAGE.
```

Figure 4.2
COBOL report program with comments

The Use of the NOTE Verb

The NOTE verb can be used to introduce explanatory comments into the body of the PROCEDURE DIVISION. The reader is given a general idea of the program's overall function and also of the nature of the functions of the logical segments.

Statement Format

```
B
↓
NOTE comment.
```

An Additional Formatting Device

In addition to the NOTE verb, notice the blank lines, or gaps, that appear between certain statements within the PROCEDURE DIVISION. The purpose of these blank lines, which occur between the sections of the PROCEDURE DIVISION, is to further partition the program into more easily understood segments. The intent of this partitioning is that the program is more easily read and understood by the reader.

Most COBOL compilers feature a mechanism that permits the insertion of blank lines through the use of compiler directive commands. These are included within the program deck along with other COBOL statements. You might be tempted to use blank records instead; however, many compilers will issue a warning message whenever they encounter blank records within COBOL program decks.

Figure 4.3
Compiler directives for source formatting

Directive	Result
SKIP 1	1 blank line
SKIP 2	2 blank lines
SKIP 3	3 blank lines
EJECT	skip to the top of the next page

Figure 4.4
Program deck with comments and directives

Page 1 of 2

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. RPT0403.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  RECORD-IMAGE        PICTURE IS X(80).
       PROCEDURE DIVISION.
       MAIN-CONTROL SECTION.
       M-C-001.
           NOTE     THIS SECTION OF THE PROGRAM SERVES TO
                    CONTROL THE PRINT AND READ PUMP SECTION
                    THAT FOLLOWS; TO START THE PUMP ONE
                    RECORD IMAGE IS READ AND THEN THE PUMP
                    IS EXECUTED; WHEN THE BLANK RECORD IS
                    FOUND, THE PUMP TERMINATES ALONG WITH
                    PROGRAM EXECUTION.
           ACCEPT RECORD-IMAGE.
           PERFORM PRINT-AND-READ-PROCESS
                UNTIL
                    RECORD-IMAGE = SPACES.
           STOP RUN.
           SKIP2
       PRINT-AND-READ-PROCESS SECTION.
       P-A-R-P-001.
```

Page 2 of 2

```
           NOTE     THIS SECTION REPRESENTS THE PRINT AND READ
                    PUMP MENTIONED EARLIER; WHEN INVOKED IT
                    PRINTS THE PREVIOUSLY READ RECORD IMAGE
                    AND THEN READS YET ANOTHER.
           DISPLAY RECORD-IMAGE.
           ACCEPT RECORD-IMAGE.
```

The IBM and WATBOL COBOL compilers permit the following compiler directives to be placed anywhere within a COBOL program to generate blank lines in the printed version of the COBOL source program. These commands are shown in figure 4.3.

Statement Format

```
B
↓
SKIP1
SKIP2
SKIP3
EJECT
```

Figure 4.5
COBOL program with revised NOTEs

Page 1 of 2

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. RPT0404.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  RECORD-IMAGE       PICTURE IS X(80).
       PROCEDURE DIVISION.
       MAIN-CONTROL SECTION.
           NOTE    THIS SECTION OF THE PROGRAM SERVES TO
                   CONTROL THE PRINT AND READ PUMP SECTION
                   THAT FOLLOWS. TO START THE PUMP ONE
                   RECORD IMAGE IS READ AND THEN THE PUMP
                   IS EXECUTED. WHEN THE BLANK RECORD IS
                   FOUND, THE PUMP TERMINATES ALONG WITH
                   PROGRAM EXECUTION.
       M-C-001.
           ACCEPT RECORD-IMAGE.
           PERFORM PRINT-AND-READ-PROCESS
               UNTIL
                   RECORD-IMAGE = SPACES.
           STOP RUN.
           SKIP2
       PRINT-AND-READ-PROCESS SECTION.
           NOTE    THIS SECTION REPRESENTS THE PRINT AND READ
```

Page 2 of 2

```
                   PUMP MENTIONED EARLIER. WHEN INVOKED IT
                   PRINTS THE PREVIOUSLY READ RECORD IMAGE
                   AND THEN READS YET ANOTHER.
       P-A-R-P-001.
           DISPLAY RECORD-IMAGE.
           ACCEPT RECORD-IMAGE.
```

When you use one of the commands in figure 4.3, you must remember several simple guidelines. First these are compiler directives; they are not COBOL statements. As a result, they are not followed by a period. They can only appear within the B-Margin. They must, however, be the only statements that appear on the record. The entire program record deck—compiler directives included—that produces the program listing shown earlier is found in figure 4.4.

The NOTE verb, while an extremely powerful commenting facility within COBOL, can also be a most dangerous one if improperly used. The most recent version of the report program discussed in this chapter will not work as anticipated because the NOTE verb is misused. Why? After all, this same report program worked satisfactorily in chapter 2. The only modification has been to include the NOTE verb.

When the NOTE verb appears as the first statement within a paragraph, the entire paragraph is interpreted by the COBOL compiler as containing only comments. Even the executable statements within each paragraph (i.e., ACCEPT, PERFORM, STOP, DISPLAY) are treated as comments. Consequently, our program now contains no executable statements! An entire working program is rendered totally useless by placing comments in the wrong location within the PROCEDURE DIVISION. To avoid this problem, the NOTE verb and its related comments should precede the paragraph to which they refer, as shown in figure 4.5.

The rules governing the structure of the NOTE verb are quite simple, however. The NOTE verb must be contained within the B-Margin of the source statement, and the only period that may appear as part of the NOTE must appear at the end of the comment. The NOTE is terminated by the first period encountered by the COBOL compiler, wherever that period happens to be. As a result, comments consisting of more than one sentence must contain only one period after the very last sentence. All of the other sentences in that comment must be terminated by a punctuation mark other than a period. A semicolon is suggested. Figure 4.5 illustrates the suggested use of the NOTE verb.

An Alternative to the NOTE Verb

A more recent COBOL standard dictated by the American National Standards Institute includes another facility for introducing comments into COBOL programs. This facility eliminates the use of the NOTE verb and its associated hazards, and permits comments to be introduced anywhere within the four divisions of a COBOL program. This is done by dedicating an entire record within the program deck to be a comment record. Comment records are records that contain an asterisk in column 7 of the record itself. When the COBOL compiler encounters an asterisk in column 7, that record is not processed further by the compiler. However, the record's contents are printed on the source listing exactly as the programmer placed them onto the COBOL source statement. Figure 4.6 shows the 80–80 listing program with asterisk comment records rather than NOTE verbs.

Statement Format

```
7
↓
*comment
```

Figure 4.6
Use of asterisk comments

Page 1 of 2

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. RPT0405.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  RECORD-IMAGE        PICTURE IS X(80).
       PROCEDURE DIVISION.
       MAIN-CONTROL SECTION.
       M-C-001.
      *    ****    THIS SECTION OF THE PROGRAM SERVES TO CONTROL
      *     THE PRINT AND READ PUMP SECTION THAT FOLLOWS.  TO
      *     START THE PUMP ONE RECORD IMAGE IS READ AND THEN
      *     PUMP IS EXECUTED.  WHEN THE BLANK RECORD IS FOUND,
      *     THE PUMP TERMINATES ALONG WITH THIS PROGRAM'S
      *     EXECUTION.
            ACCEPT RECORD-IMAGE.
            PERFORM PRINT-AND-READ-PROCESS
                UNTIL
                    RECORD-IMAGE = SPACES.
            STOP RUN.
            SKIP2
       PRINT-AND-READ-PROCESS SECTION.
       P-A-R-P-001.
      *    ****    THIS SECTION REPRESENTS THE PRINT AND READ
```

Page 2 of 2

```
      *    PUMP MENTIONED EARLIER.  WHEN INVOKED, IT PRINTS
      *    THE PREVIOUSLY READ RECORD IMAGE AND THEN READS
      *    YET ANOTHER.
           DISPLAY RECORD-IMAGE.
           ACCEPT RECORD-IMAGE.
```

Unlike the NOTE verb, shown in figures 4.2 and 4.4, there is no danger of an entire paragraph's statements being interpreted as comments. The only comments are those statements that appear on the record with an asterisk in column 7. Also, these comment records can, unlike the NOTE verb, appear anywhere within a COBOL program. The NOTE verb can only be used within the PROCEDURE DIVISION. Future programming examples illustrated in this text will utilize the asterisk method of commenting. Also, compiler directives will include the use of blank lines for partitioning the program into its logical segments.

Summary

COBOL programs written without the use of comments are of little value to an organization. Without comments, COBOL programs are likely to be difficult to read, comprehend, develop, and maintain.

Students learning the COBOL language are usually optimists. They believe that they will have little difficulty in completing their programming assignments. They are unlikely to see the value of including comments in a COBOL program as the program is being constructed. Many reasons can be cited for this attitude. Students may feel that comments take too much time to write and key, or that they intend to add them when the program has been completed. What these students are really saying, however, is that their programming skills are so good that they do not need to use comments while their programs are being created. However, this is not true.

Poor Programming Technique
The programmer who adds commentary to a program only after the program functions properly.

How often has the following happened to you? While in the midst of writing a term paper you are stopped by a phone call, an appointment, a class, or a similar interruption. You later return to your writing and review what you have already written. The review, however, causes you some concern. You cannot remember the line of reasoning you were using prior to the interruption. Or, perhaps, you are no longer pleased with what you did earlier. The point is, you must be able to review your earlier efforts and understand your logic.

In the case of a term paper you would probably have little difficulty in picking up where you had stopped. The reasons include the fact that you probably think, read, and write in English all of the time. No one thinks, reads, or writes in COBOL all of the time.

Comments in COBOL programs are written in English, not in COBOL. The comments summarize the purpose of a program or one of its logical sections. These comments are written in your words to aid you and anyone else who is likely to read your program.

These points concerning commenting are seldom taken to heart by the beginning student. The return on the effort required to construct comments just isn't obvious at first. It does become obvious, however, when the student moves from the classroom to the programmer's bench; particularly because very few programming jobs exclude the task of program maintenance.

Good Programming Technique
The programmer who adds commentary as the program is being constructed and realizes that this effort is beneficial.

Program maintenance almost always forces the programmer to alter or repair a computer program written by another programmer. Unfamiliar programs without clear, concise comments are very difficult to understand, and thus to maintain. Furthermore, maintenance is usually conducted under tense time constraints. A malfunctioning payroll program must be working in time to produce the company's payroll. Requests for such repairs can come during the middle of the night before the paychecks are scheduled to be delivered to the organization's staff. This is unlikey to be a leisurely repair mission. The program you fix has to be readable. Comments will help to guide you through the maze of logic pathways that exist within the program.

Questions for Review

1. What is meant by the phrase that COBOL is a self-documenting language? Is this statement valid? Why?
2. At what points in a COBOL program are comments to be placed? What should those comments describe?
3. Of the two options discussed in this chapter, which method of introducing comments offers the greatest flexibility with the least possible chance of error? The NOTE verb? The asterisk comment record? Why?
4. Is the following statement true? Why?

   Comment records created by using an asterisk in column 7 of the record image should not immediately follow a paragraph name in the PROCEDURE DIVISION of a COBOL program. To do so renders all of the executable statements within that paragraph inoperative.
5. If SKIP1, SKIP2, and SKIP3 can introduce blank lines to separate logical segments of a COBOL program, then what would be a possible usage of EJECT? (Hint: Are bigger separating gaps better than smaller ones?)

Problems

1. Examine the program without comments shown in figure 4.1. Assume that the function of this 80–80 listing program is to successively read and print sales invoices for a plumbing supply firm. Reconstruct the program with the comments appropriate to the program's task. Use asterisk comment cards.
2. Repeat the task in exercise 1. This time, however, use the NOTE verb rather than the asterisk comment card.

Programming Assignments

1. Revise one (or both) of the programming assignments from chapter 3 so that:
   a. each major logical unit of the PROCEDURE DIVISION is properly commented, and
   b. each division and section is physically isolated by blank (horizontal) lines.

Report Titles and Trailers
Part 1

5

Chapter Outline

The report program created in the previous chapters performs no more than an 80–80 listing function. In fact, many items you would normally expect to find on any report appear to be missing. An examination of the pseudo-coded version of our report program in figure 5.1 shows that no provisions for report *titles, trailers,* and the like have been made. Given that the program logic shown in figure 5.1 represents the structure of any report program, where should the instructions to print report titles be placed? Further, what titles should be printed?

Titles
The printed lines that appear at the top of each page of the printed report.

Trailers
The printed lines that appear at the bottom of each page of the printed report.

Assume that the report program being designed will print the payroll report for a hypothetical company. This report will appear as shown in figure 5.2. The report will provide the appropriate corporate executives with a list of all the divisions' employees involved with the sale of recreation-related products. Along with the name and social security number of each employee, management will be able to see how many hours each employee worked during the last payroll period as well as the employee's hourly wage. Where, in the pseudo-coded version of our report program logic shown in figure 5.1, should the DISPLAY statements be placed in order to produce these titles and column headings? They must be printed at the beginning of the payroll report.

Figure 5.1
Report program

```
Statement       Pseudo-coded statement
 number

   1            read very first payroll record image.
   2            do the print and read process
                     until
                          record read is blank.
   3            stop execution.

   4       print and read process
   5            print that very same record image.
   6            read another payroll record image
```

Figure 5.2
Sample payroll report with titles and trailers

```
                 ACME MACHINE COMPANY
          LEISURE AND SPORTS PRODUCTS DIVISION
                MINNEAPOLIS SALES OFFICE

                 WEEKLY PAYROLL REPORT


         SOCIAL        EMPLOYEES       HR    HR
         SECURITY      NAME            WK    WG
         NUMBER

         215-50-3145   WEINER, HD      18    300
         197-42-1508   BAGGINS, FX     25    275
              .             .           .     .
              .             .           .     .
              .             .           .     .

                 COMPANY CONFIDENTIAL
```

Producing Report Titles

Our pseudo-coded version of the report program has been modified to illustrate the proper location for printing report titles. As figure 5.3 reveals, the ideal place to print report titles is at the very beginning of the program. In this manner you avoid misplacing the instructions, which could cause the titles to be printed in the wrong place.

To understand the importance of proper placement, we can see what happens when we insert the instruction to print the report titles in the wrong place.

What if statement 0 is placed after statement 2 instead? You should recall that statement 2 is the iterative statement that causes successive input records to be read and printed. Placement of statement 0 after this statement has a rather unusual effect. Remember, control passes from statement 2 to statement 3 only if the last input record read is the terminator record. The terminator record must be the last record in the input file. As a result, the titles will be printed at the end of the report. Clearly, this is far from satisfactory.

What if the statement to print titles is placed after statement 3? Statement 3 terminates the execution of our report program. After this statement completes its function the computer has nothing more to do with the program. Consequently, control could never pass from statement 3 to the statement to produce our titles. This too is unsatisfactory.

Could statement 0 be placed anywhere within the confines of the group named *print-and-read-process?* No. To do so would mean that every time this group is activated by the iterative statement the titles would be printed. The result would be that the report titles would appear before each and every input record image printed in the report. If the group was entered 100 times, then the titles would be printed 100 times! We want the titles only at the beginning of the report.

```
Statement       Pseudo-coded statement
 number

   0            print report titles
   1            read very first payroll record read.
   2            do the print and read process
                      until
                            record read is blank.
   3            stop execution.

   4      print and read process
   5            print that very same record image.
   6            read another payroll record image
```

Figure 5.3
Program logic to include titles

Figure 5.4
Logic modified to print trailers

```
Statement    Pseudo-coded statement
 number

 0                print titles
 1                read very first payroll record read.
 2                do the print and read process
                        until
                              record read is blank.
 2.1              print report trailer
 3                stop execution.

 4           print and read process
 5                print that very same record image.
 6                read another payroll record image
```

Producing Report Trailers

Not all reports contain trailers—or messages—at the bottom of each printed page, but many do. Since our report requires the use of the trailer COMPANY CONFIDENTIAL, we must modify the pseudo-coded program to account for this operation.

We just discussed the impact of improperly placing the statement to print report titles. One location mentioned would have caused our titles to be printed only after all of the input records had been read and printed. That particular location was immediately after statement 2, the iterative statement. This is the proper place to add a statement to print a report trailer. We shall modify the program logic once again to print a report trailer. Our modified program logic shown in figure 5.4 is now fully capable of producing the facsimile report, complete with its associated report titles and trailer.

Actually Printing Titles and Trailers

Our program design has now been altered to print titles and trailers. Step one of the programming process, developing the logic, is done. Step two involves translating each pseudo-coded statement into its COBOL equivalent. Report titles and trailers are printed by using COBOL output operations.

The only COBOL instruction we have covered that enables us to print anything is the DISPLAY instruction. Not only do we DISPLAY input records read, but we also DISPLAY report titles and trailers. To learn how to do this we must review the titles that are expected to be present in our computer-produced report. The ten lines shown in figure 5.5 include separating blank lines as well as those lines that contain real printed material. A blank line separates the corporate identification and the name of the report. Two blank lines separate the name of the report from the three-line column headings.

The DISPLAY instruction prints the contents of the variable whose name follows the verb. In any event, each invocation of the instruction causes only one line to be printed. So, for ten lines of titles, we must have ten DISPLAY instructions. It also stands to reason that we need to define ten variables in the DATA DIVISION. Each variable corresponds to one line of the titles—ten lines require ten variables.

Our program logic reflects only one statement dedicated to printing titles. Obviously, many more actual COBOL instructions are required. The PROCEDURE DIVISION of our COBOL program is likely to look like that shown in figure 5.6. It has started to grow in size rather dramatically.

Inspection of the PROCEDURE DIVISION just listed shows that actually ten variables are not required. Only eight variables are necessary. This is because the same blank line can be printed wherever a blank line is required. A separate definition of each blank line is not necessary. What do the variables TITLE-1 thru TITLE-7 look like? TITLE-1 corresponds to the first line of the title. That line, like all of the lines printed in this report, is 80 characters in length.

Figure 5.5
Titles expected in payroll report

```
                    ACME MACHINE COMPANY
            LEISURE AND SPORTS PRODUCTS DIVISION
                 MINNEAPOLIS SALES OFFICE

                   WEEKLY PAYROLL REPORT


     SOCIAL             EMPLOYEES             HR       HR
     SECURITY           NAME                  WK       WG
     NUMBER
```

Figure 5.6
PROCEDURE DIVISION that also prints titles

Page 1 of 2

```
       PROCEDURE DIVISION.
       MAIN-CONTROL SECTION.
       M-C-001.
      *    ****    PRINT THE REPORT TITLES THAT MUST APPEAR AT THE
      *            BEGINNING OF THE REPORT.
           DISPLAY TITLE-1.
           DISPLAY TITLE-2.
           DISPLAY TITLE-3.
           DISPLAY BLANK-LINE.
           DISPLAY TITLE-4.
           DISPLAY BLANK-LINE.
           DISPLAY BLANK-LINE.
           DISPLAY TITLE-5.
           DISPLAY TITLE-6.
           DISPLAY TITLE-7.
           SKIP2
       M-C-002.
      *    ****    THIS SECTION OF THE PROGRAM SERVES TO CONTROL THE
      *            PRINT AND READ PUMP SECTION THAT FOLLOWS.  TO
      *            START THE PUMP ONE RECORD IMAGE IS READ AND THEN
      *            PUMP IS EXECUTED.  WHEN THE BLANK RECORD IS FOUND,
      *            THE PUMP TERMINATES ALONG WITH THIS PROGRAM'S
      *            EXECUTION.
           ACCEPT RECORD-IMAGE.
```

Page 2 of 2

```
           PERFORM PRINT-AND-READ-PROCESS
               UNTIL
                   RECORD-IMAGE = SPACES.
           STOP RUN.
           SKIP2
       PRINT-AND-READ-PROCESS SECTION.
       P-A-R-P-001.
      *    ****    THIS SECTION REPRESENTS THE PRINT AND READ PUMP
      *            MENTIONED EARLIER.  WHEN INVOKED, IT PRINTS THE
      *            PREVIOUSLY READ RECORD IMAGE AND THEN READS YET
      *            ANOTHER.
           DISPLAY RECORD-IMAGE.
           ACCEPT RECORD-IMAGE.
```

Initialized Variables

The only variable that we have defined thus far is RECORD-IMAGE. That declaration is shown in figure 5.7. As defined here, RECORD-IMAGE is said to be uninitialized. That is, until our first ACCEPT instruction fills RECORD-IMAGE with an input record, we have no idea whatsoever as to the contetns of RECORD-IMAGE. Uninitialized variables are said to have indeterminate contents.

This is not so in the definition shown in figure 5.8. Here BLANK-LINE has been defined as an initialized variable. We know precisely the contents of this particular variable before our program begins to execute. Each of the 80 positions in BLANK-LINE is said to be blank. When this variable is DISPLAYed, a blank line will appear on the printer.

Statement Format

```
A    B
↓    ↓
01   variablename   PICTURE IS attributes VALUE non-numeric
                                                    literal.
```

To create initialized variables, the programmer needs to add the VALUE attribute specification to the variable definition. As has always been the case, the entire declaration is terminated by a period. Note how at least one blank is used to separate each of the parts of the declaration.

TITLE-1 must be an initialized variable, such as BLANK-LINE. Unlike BLANK-LINE, however, this variable does not contain 80 blanks. Instead, TITLE-1 holds the first line of the ten-line title originally displayed in figure 5.5. The declaration of TITLE-1 is shown in figure 5.9. Unlike the word SPACES used in BLANK-LINE, our initialization specification is a nonnumeric literal. *Nonnumeric literals* can contain any alphanumeric character including blanks. These literals are said to be *delimited* by quotes. In other words, the beginning and end of the literal are marked by the quotes. The quotes are not part of the literal itself. Only the characters surrounded by the quotes are part of the literal.

Nonnumeric Literal
A collection or sequence of values that is not or cannot be manipulated in an arithmetic operation.

Delimiter
A special character that marks both the start and the end of nonnumeric literals. In COBOL this character is a quote or an apostrophe.

Compare the length of the nonnumeric literal and the length of the field into which the literal is to be placed. The literal is 20 characters long while the field itself is 80. When the literal is smaller than its containing field, a question arises. Where, within the 80 characters, is the 20 character literal to be placed by the COBOL compiler? In alphanumeric variables the literal is placed in the left-hand position of the field. Thus, this literal occupies the first 20 of the 80 characters. The remaining 60 characters are automatically set to blanks by the COBOL compiler.

Figure 5.7
Sample variable definition

```
       01  RECORD-IMAGE      PICTURE IS X(80).
```

Figure 5.8
Initialized COBOL variable

```
       01  BLANK-LINE            PICTURE IS X(80) VALUE SPACES.
```

Figure 5.9
TITLE-1—another initialized variable

```
       01  TITLE-1           PICTURE IS X(80) VALUE "ACME MACHINE COMPANY".
```

Unfortunately, TITLE-1 is not defined properly. If its contents were displayed, the title would not be centered within the first line of the title. Instead, the phrase ACME MACHINE COMPANY would appear in the left-hand position of the printed page. That is, the nonblank portion of the title, because it was automatically placed into the extreme left-hand side of the field, will also print *left justified.* To overcome this problem we must alter our original definition. Figure 5.10 illustrates how the definition should appear. Our declaration now requires two source statements because it is impossible to fit the entire declaration onto a single source statement and still remain within the B-Margin. Now the letter *A* in ACME is preceded by 30 blanks. The phrase ACME MACHINE COMPANY will be centered on our report when it is displayed because it is also centered within the field.

Left Justification
When values are moved to nonnumeric fields they are started in the left-hand position of those fields.

Note that the entire length of the literal is now 50 characters. The field remains at 80 characters. Thus the literal is said to occupy the first 50 of the 80 characters, while the remaining 30 characters are blank positions that have been automatically set by the COBOL compiler.

The next several title variables are defined in the same manner as TITLE-1. First, the number of characters in each title phrase is counted. In the case of the second title phrase, LEISURE AND SPORTS PRODUCTS DIVISION, the number of characters is 36. Since the print line is 80 characters long, a total of 44 blanks are in this title. If the title is to be centered, then 22 of these blanks must be included on either side of the phrase LEISURE AND SPORTS PRODUCTS DIVISION, as depicted in figure 5.11. This particular nonnumeric literal ends in column 70. The delimiting quote and period occupy 2 additional columns. Column 72, you should remember, is the last column of the B-Margin. What would have happened if the phrase to be centered was larger than LEISURE AND SPORTS PRODUCTS DIVISION? You would be required to continue the nonnumeric literal on still another source statement.

The next two declarations, shown in figure 5.12, must also have their phrases centered. For practice, determine the number of blanks required to surround both phrases in order to center each phrase. Then verify the correctness of these next declarations.

When a nonnumeric literal is placed on its own source statement, the literal must reside wholly within the bounds of the B-Margin. This includes the delimiting quotes and the period, too. The B-Margin starts in column 12 and ends in column 72, so the B-Margin is composed of 61 columns. Three of those columns are reserved for two quotes and the period. Thus, only 58 columns are actually available for use by the nonnumeric literal if the entire literal is to be located on just one source statement.

```
       01  TITLE-1          PICTURE IS X(80) VALUE
           "                              ACME MACHINE COMPANY".
```

Figure 5.10
TITLE-1—centered initial value

```
       01  TITLE-2          PICTURE IS X(80) VALUE
           "                      LEISURE AND SPORTS PRODUCTS DIVISION".
```

Figure 5.11
Centering the second report title

```
       01  TITLE-3          PICTURE IS X(80) VALUE
           "                         MINNEAPOLIS SALES OFFICE".
       01  TITLE-4          PICTURE IS X(80) VALUE
           "                            WEEKLY PAYROLL REPORT".
```

Figure 5.12
Definitions of TITLE-2, TITLE-3, and TITLE-4

Figure 5.13
Titles in payroll report

```
                ACME MACHINE COMPANY
        LEISURE AND SPORTS PRODUCTS DIVISION
             MINNEAPOLIS SALES OFFICE

               WEEKLY PAYROLL REPORT


SOCIAL            EMPLOYEES             HR      HR
SECURITY          NAME                  WK      WG
NUMBER
```

Examine the last three title lines in our payroll report. They appear in figure 5.13. Each of the three titles is longer than 58 columns if the trailing blanks are counted. The three variable definitions in figure 5.14 complete all of our report title variables. Of course TITLE-5, TITLE-6, and TITLE-7 could have had their nonnumeric literals defined using only one source statement. This could have been accomplished by terminating the literals after the last nonblank character on each of the three declarations. Then the literals would have been smaller than the variable. As you know, when this situation occurs, the COBOL compiler automatically fills the variable with blanks to the right of the literal.

What we have instead is the continuation of a nonnumeric literal. When continued onto a second source statement, a nonnumeric literal

1. is not terminated by a quote or a period on the first of its two source statements. In the first source statement, it is assumed that the literal extends through column 72;
2. is marked with a hyphen in column 7 of the subsequent statement to show continuation. As we discussed earlier, COBOL statements can be continued onto more than one source statement without the use of any continuation character. Only nonnumeric literals, when continued onto a second source statement, require the use of a continuation character in column 7 of the subsequent source statement. Also, the continued portion of the nonnumeric literal is again started with a beginning quote somewhere in the confines of the B-Margin;
3. must have a closing quote and period specified on the last source statement required to hold a continued nonnumeric literal.

Now that all of our title variables have been defined, the WORKING-STORAGE SECTION of the DATA DIVISION of our COBOL program takes on the appearance illustrated in figure 5.15.

Figure 5.14
Remaining title definitions

```
       01  TITLE-5           PICTURE IS X(80) VALUE
           "SOCIAL                EMPLOYEES               HR          HR
      -    "                       ".
       01  TITLE-6           PICTURE IS X(80) VALUE
           "SECURITY              NAME                    WK          WG
      -    "                       ".
       01  TITLE-7           PICTURE IS X(80) VALUE
           "NUMBER
      -    "                       ".
```

Figure 5.15
Complete WORKING-STORAGE SECTION

Page 1 of 2

```
       DATA DIVISION.
       WORKING-STORAGE SECTION.
      *    ****    ALL REPORT TITLES AND TRAILERS
      *
       01  TITLE-1           PICTURE IS X(80) VALUE
           "                                ACME MACHINE COMPANY".
       01  TITLE-2           PICTURE IS X(80) VALUE
           "                         LEISURE AND SPORTS PRODUCTS DIVISION".
       01  TITLE-3           PICTURE IS X(80) VALUE
           "                              MINNEAPOLIS SALES OFFICE".
       01  TITLE-4           PICTURE IS X(80) VALUE
           "                               WEEKLY PAYROLL REPORT".
       01  TITLE-5           PICTURE IS X(80) VALUE
           "SOCIAL                EMPLOYEES               HR          HR
      -    "                       ".
       01  TITLE-6           PICTURE IS X(80) VALUE
           "SECURITY              NAME                    WK          WG
      -    "                       ".
       01  TITLE-7           PICTURE IS X(80) VALUE
           "NUMBER
      -    "                       ".
       01  BLANK-LINE        PICTURE IS X(80) VALUE SPACES.
           SKIP2
      *    ****    ALL OTHER VARIABLES
```

Page 2 of 2

```
      *
       01  RECORD-IMAGE PICTURE IS X(80).
```

Summary

All reports require titles and often trailers, too. These titles and trailers must be defined as variables in the WORKING-STORAGE SECTION of the COBOL program. Unlike earlier declarations, which were said to be uninitialized variables, these new variables must be given initial values. Those values correspond to each of the lines in the title or trailer messages to be printed.

Initial values are specified by the use of the VALUE attribute followed by a nonnumeric literal. These literals are delimited by quotes and the entire definition is then terminated by a period.

When the literal is smaller than the variable in which it resides, the COBOL compiler places the literal in the left-hand positions within the variable. This is called left justification. Whatever unused positions exist on the right-hand side of the variable are automatically set as blanks.

Because of left justification of literals in alphanumeric fields, items to be centered on a print line pose a special problem to the COBOL programmer. The programmer must determine the number of blanks that are to precede the centered phrase and then begin the nonnumeric literal with that same number of blanks.

Nonnumeric literals that are longer than 58 positions are likely to require continuation onto a second source statement. Great care should be taken to avoid terminating the first portion of that literal with a quote and a period. Instead, the literal must extend through column 72. The continued portion of the literal must be marked with a hyphen in column 7. The continued literal resumes with a quote in the B-Margin and is ended with a quote and a period.

Our entire COBOL program can be found in figure 5.16.

Figure 5.16
Completed COBOL program with titles

Page 1 of 3

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.    RPT0501.
           SKIP2
       ENVIRONMENT DIVISION.
           SKIP2
       DATA DIVISION.
       WORKING-STORAGE SECTION.
      *    ****    ALL REPORT TITLES AND TRAILERS
      *
       01  TITLE-1             PICTURE IS X(80) VALUE
           "                              ACME MACHINE COMPANY".
       01  TITLE-2             PICTURE IS X(80) VALUE
           "                      LEISURE AND SPORTS PRODUCTS DIVISION".
       01  TITLE-3             PICTURE IS X(80) VALUE
           "                                    MINNEAPOLIS SALES OFFICE".
       01  TITLE-4             PICTURE IS X(80) VALUE
           "                                    WEEKLY PAYROLL REPORT".
       01  TITLE-5             PICTURE IS X(80) VALUE
           "SOCIAL              EMPLOYEES             HR          HR
      -    "                   ".
       01  TITLE-6             PICTURE IS X(80) VALUE
           "SECURITY            NAME                  WK          WG
      -    "                   ".
       01  TITLE-7             PICTURE IS X(80) VALUE
```

Questions for Review

1. Why should the COBOL programmer—during program construction—take great care to specify initial values for variables at compilation rather than execution time?
2. In assigning an initial value to an alphanumeric field, what happens if the literal is
   a. larger than the field?
   b. the same size as the field?
   c. smaller than the field?
3. The nonblank portion of a nonnumeric literal is to be centered within the variable to which it is assigned. What steps must the programmer take?
4. Why do nonnumeric literals become a problem when they exceed 58 characters? How is the problem solved?
5. Why must nonnumeric literals be delimited by a quote?

Page 2 of 3

```
           "NUMBER
      -    "                         ".
       01  TRAILER-1      PICTURE IS X(80) VALUE
      -    "                              COMPANY CONFIDENTIAL".
       01  BLANK-LINE     PICTURE IS X(80) VALUE SPACES.
           SKIP2
      *    ****  ALL OTHER VARIABLES
      *
       01  RECORD-IMAGE PICTURE IS X(80).
           SKIP2
       PROCEDURE DIVISION.
       MAIN-CONTROL SECTION.
       M-C-001.
      *    ****     PRINT THE REPORT TITLES THAT MUST APPEAR AT THE
      *             BEGINNING OF THE REPORT.
           DISPLAY TITLE-1.
           DISPLAY TITLE-2.
           DISPLAY TITLE-3.
           DISPLAY BLANK-LINE.
           DISPLAY TITLE-4.
           DISPLAY BLANK-LINE.
           DISPLAY BLANK-LINE.
           DISPLAY TITLE-5.
           DISPLAY TITLE-6.
```

Page 3 of 3

```
           DISPLAY TITLE-7.
           SKIP2
       M-C-002.
      *    ****      THIS SECTION OF THE PROGRAM SERVES TO CONTROL THE
      *              PRINT AND READ PUMP SECTION THAT FOLLOWS.  TO
      *              START THE PUMP ONE RECORD IMAGE IS READ AND THEN
      *              PUMP IS EXECUTED.  WHEN THE BLANK RECORD IS FOUND,
      *              THE PUMP TERMINATES ALONG WITH THIS PROGRAM'S
      *              EXECUTION.
           ACCEPT RECORD-IMAGE.
           PERFORM PRINT-AND-READ-PROCESS
                   UNTIL
                         RECORD-IMAGE = SPACES.
           DISPLAY TRAILER-1.
           STOP RUN.
           SKIP2
       PRINT-AND-READ-PROCESS SECTION.
       P-A-R-P-001.
      *    ****      THIS SECTION REPRESENTS THE PRINT AND READ PUMP
      *              MENTIONED EARLIER.  WHEN INVOKED, IT PRINTS THE
      *              PREVIOUSLY READ RECORD IMAGE AND THEN READS YET
      *              ANOTHER.
           DISPLAY RECORD-IMAGE.
           ACCEPT RECORD-IMAGE.
```

Problems

1. Examine the following report program logic as expressed in pseudo-coding. What, if anything, is wrong with this logic?

```
          read very first input record
          do print and read process
               until
                    record read is blank
          stop run
          print report trailers
     print and read process
          print record read
          read next input record
          print titles
```

   When will the report titles be printed? Report trailers? Why?

2. Using the dotted guide as an aid, define the title variables that would be required to produce the following report titles.

```
.........1.........2.........3.........4.........5
          NYC CENTER FOR SOCIAL DISEASES
               SOCIAL ACTION GROUP
               COUNTER  ACTIVITIES
                    PAYROLL
```

   Assume that the length of the print line is 50 characters.

3. Examine the following variable declarations. What, if anything, is wrong?

```
01   TITLE-1    PICTURE IS X(80)
     "                 ACME MANUFACTURING COMPANY.
01   TITLE-2    PICTURE IS X(80) VALUE
     "     THIS IS THE BEGINNING OF THE TITLE
     "THIS IS ITS CONTINUATION/CONCLUSION"
01   TRAILER-1 PICTURE IS X(80) VALUE
     "  OUR TRAILER MESSAGE BEGINS HERE.
-    "BUT ENDS HERE."
```

Programming Assignments

1. Bio-Technical Assessment Laboratories employs a large number of highly skilled professional personnel who submit time cards on a weekly basis. The format of the time cards is shown below.

Record Position	*Data Type*	*Variable*
01–09	N	Social security number
10–24	AN	Employee's name
25–26	N	Hours worked
27–30	N	Hourly wage
31–32	N	Federal withholding deductions
33–34	N	State withholding deductions
35–80	—	Unused

   The labor agreement between Bio-Technical and the employees' labor union requires that a weekly report listing each time card be forwarded to the union headquarters. The format of that report can be found in figure 5.17.

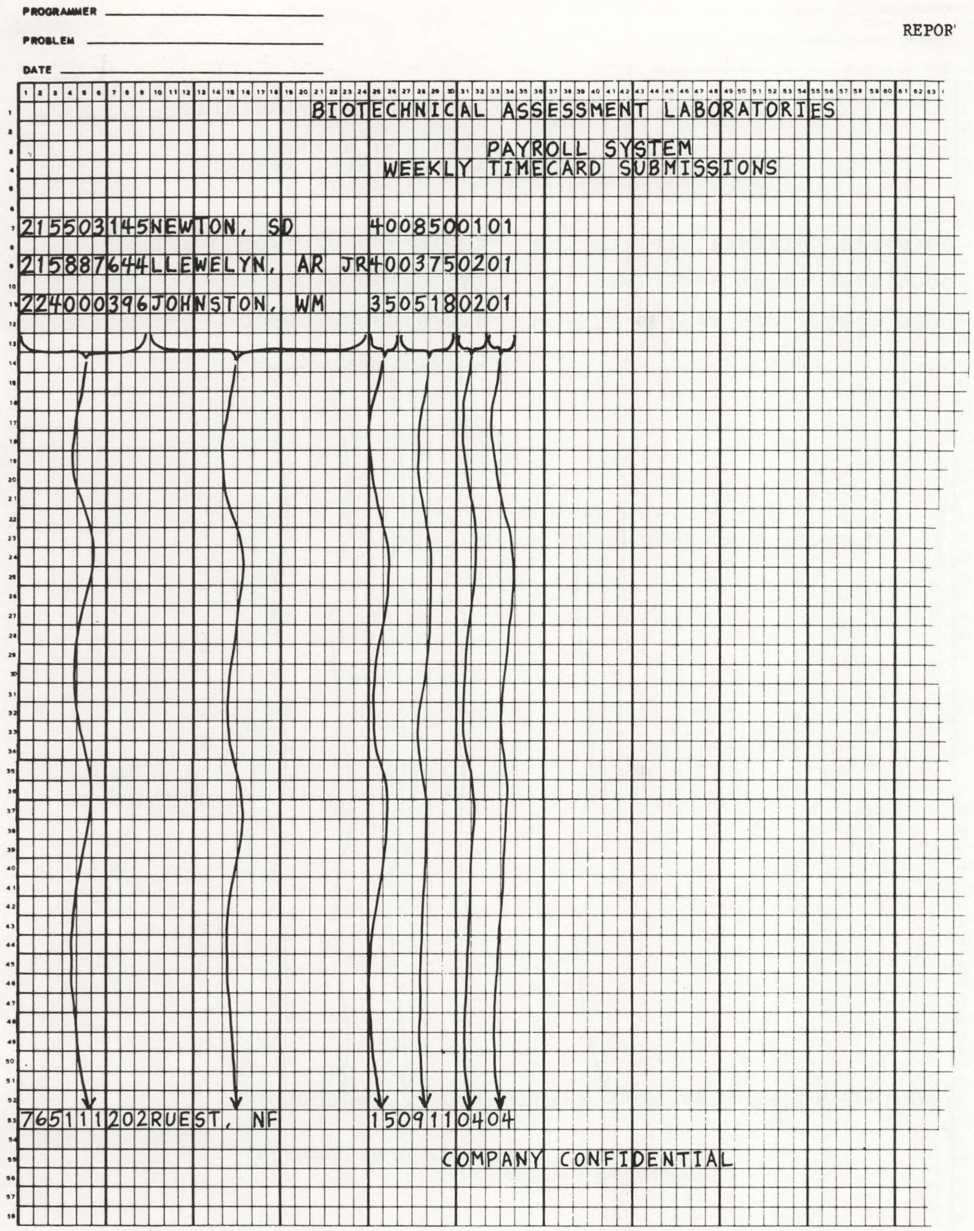


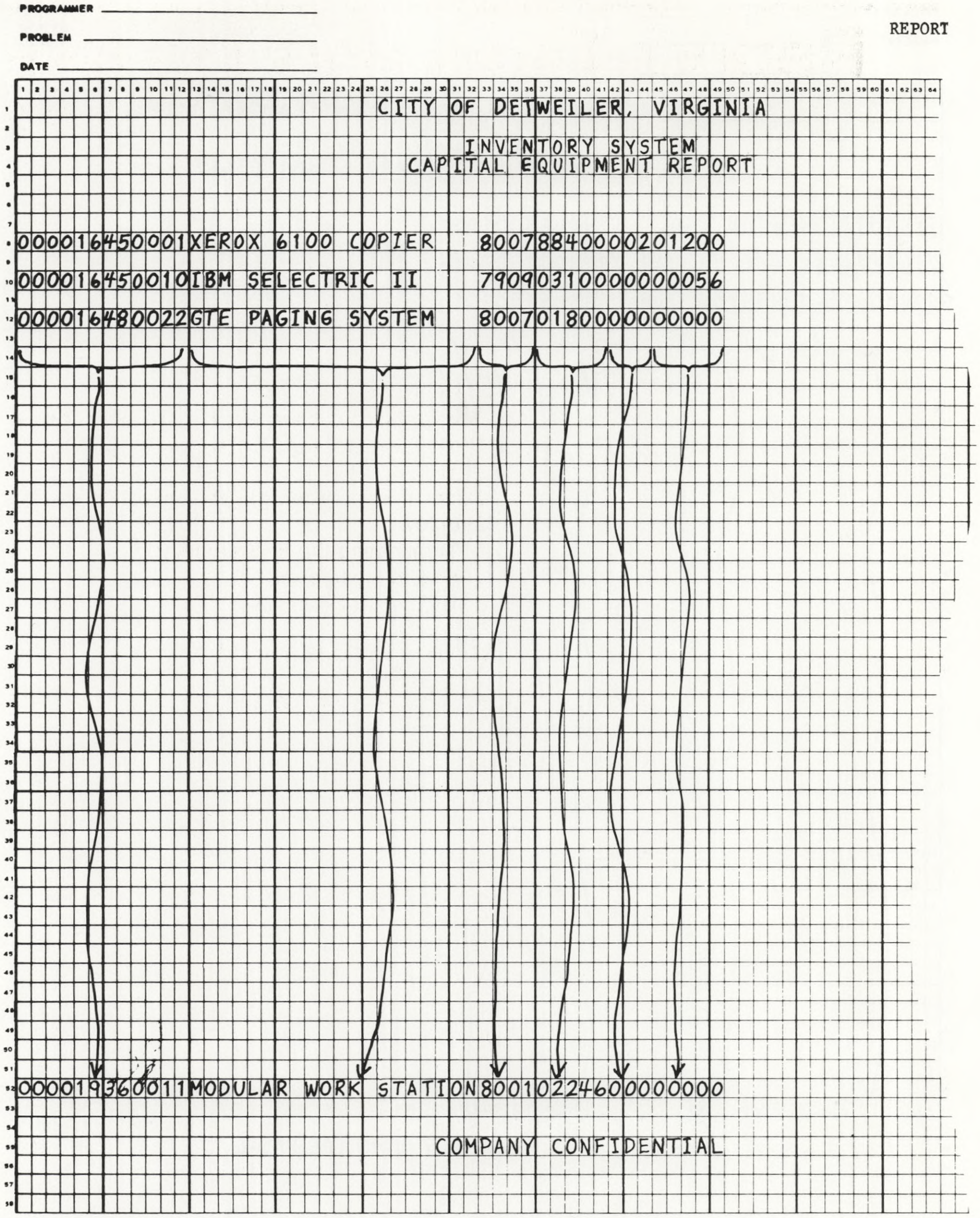
Figure 5.17
Bio-Technical Laboratories time card report

2. Virginia state law requires that each municipality file a quarterly capital equipment report listing each item of equipment owned by the municipality. To meet this requirement the city of Detweiler engaged a local accounting firm to develop a capital equipment management system. At the heart of this system is the Capital Equipment File. Each record in that file appears according to the following format:

Record Position	*Data Type*	*Variable*
01–12	AN	Equipment serial number
13–32	AN	Equipment description
33–36	N	Date of acquisition (YYMM)
37–41	N	Acquisition cost ($)
42–44	N	Days out of service
45–49	N	Repair costs ($)
50–80	—	Unused

The report to be produced can be found in figure 5.18.

Figure 5.18
Detweiler capital equipment report



Report Titles and Trailers

Part 2

6

Chapter Outline

The pseudo-coded logic introduced in chapter 2 was modified in chapter 5 to provide a limited ability to print report titles and trailers. However, the logic must be modified once more if these report titles and trailers are to appear at the top and bottom of each page printed by our COBOL program.

At present, our report program prints report titles only at the beginning of the report and prints trailers only at the conclusion. It doesn't matter whether the file of input records contains 1, 100, or 10 thousand records. In fact, a reexamination of the pseudo-code, shown in figure 6.1, shows a complete disregard for the concept of *page overflow*. Once the report titles have been printed, the print-and-read process is performed continuously, with no inquiry concerning how many lines remain to be printed on each page. This implies that the page is infinitely long, which we know is not true.

Page Overflow
A situation that occurs when printing the next record image will cause a line of printed output to appear in that portion of the page reserved for the trailers.

Another problem to be solved concerns page numbers. Multipage reports feature consecutive page numbers on each page; therefore, the report title must be modified to include the ability to number pages. To accomplish these objectives, this chapter will introduce two new concepts: (1) numeric variables and (2) a decision structure.

Numeric Variables

The first of these concepts is a new variable type—the *numeric variable.* Numeric variables, unlike alphanumeric variables, can be used in arithmetic operations. Alphanumeric variables cannot be used in arithmetic operations because they can contain nonnumeric characters, such as letters of the alphabet, special characters, and blanks. Alphanumeric variables were defined as possessing PICTURE IS X.... attributes. The basic models for numeric variable definitions appear in figure 6.2.

Numeric Variable
A COBOL field that contains only the digits 0 through 9 in each of its positions.

Uninitialized Variable
A field whose value at the start of execution is not known.

Initialized Variable
A field whose contents at the start of execution are known due to the use of the VALUE attribute in the field's definition.

The first declaration in figure 6.2 illustrates an *uninitialized variable* declaration. The value of an uninitialized variable is unknown when a program begins to execute. An attempt to involve an uninitialized numeric variable in an arithmetic operation is likely to prove disastrous. The second declaration is an example of an *initialized variable* definition. The question of how a numeric variable should be initialized is an interesting one.

Figure 6.1
Report program logic minus page overflow

```
Statement          Pseudo-coded statement
 number

   1                    print report titles
   2                    read very first inventory record image.
   3                    do the print and read process
                              until
                                    record image is blank.
   4                    print report trailers
   5                    stop execution.

   6              print and read process
   7                    print that very same record image.
   8                    read another inventory record image
```

Figure 6.2
Definition of numeric variables

```
       01  VARIABLENAME       PICTURE IS 9 (repetition factor).
```

or

```
       01  VARIABLENAME       PICTURE IS 9 (repetition factor)
                              VALUE numeric literal.
```

Initialization: Compilation versus Execution

Before any numeric variable can be used in an arithmetic statement, it must have a valid initial numeric value. A valid initial value can be assigned in one of two ways. Either the VALUE clause is used when the variable is defined, or an executable instruction must provide that initial value. The two alternatives might be viewed as shown in figure 6.3. Here the variable CURRENT-LINE is used to monitor the line of the page that is currently being printed. Proper use of this variable aids programs in determining whether the end of a page has been reached. Every time a line is DISPLAYed, our program adds one to this particular variable.

Since CURRENT-LINE is involved in an arithmetic operation, it must possess a valid numeric value prior to the start of that operation. Either this value is assigned by a VALUE clause, or it must be assigned by a verb of action in the PROCEDURE DIVISION, such as the MOVE instruction. Whenever possible, you should initialize a variable through the use of a VALUE clause. This requires far less effort than it would take to write an executable statement to accomplish the same task. If the variable is initialized when the program is compiled, it will have the same starting value whenever the program is executed. No action is required to establish the variable's beginning value. The VALUE clause does this automatically. On the other hand, if the variable is initialized through an instruction, such as MOVE, then that instruction must execute every time the program is run. If the instruction does not execute, the variable's contents are unknown, or unpredictable, because the variable was declared without a VALUE clause.

Statement Format

```
A    B
↓    ↓
01   variablename      PICTURE IS attributes VALUE
                                          numericliteral.
```

Another reason that you should initialize variables at compilation is even more important. Programmers often unintentionally place executable instructions in the wrong place within a COBOL program. It is easy to misplace a statement. Some misplacements have no undesirable impact while others can be devastating. An execution statement that provides a variable with its initial value must be executed before an arithmetic instruction attempts to use that variable. If the statement is not executed, the variable will not contain a valid beginning value. The assumption that numeric

```
       01  CURRENT-LINE      PICTURE IS 9(03)   VALUE ZERO.
```

or

```
                  MOVE ZERO TO CURRENT-LINE.
```

Figure 6.3
Alternatives in initializing numeric variables

Figure 6.4
Use of uninitialized variables

```
       01  CURRENT-LINE        PICTURE IS 9(03).
           .
           .
           .
           ADD 1 TO CURRENT-LINE.
           MOVE 0 TO CURRENT-LINE.
```

fields contain numeric values will not be true in this case. The arithmetic instruction will fail, and the program will abnormally terminate. That is quite a large price to pay, as indicated in figure 6.4. The program from which these excerpts were taken would unexpectedly terminate when the computer attempted to execute the ADD instruction. The reason: CURRENT-LINE does not contain a valid numeric value. The MOVE instruction is listed one instruction too late.

Statement Format

```
B
↓
ADD numericliteral TO variablename.
```

Alternatives in Compilation Initialization

Numeric Literals
A sequence of characters whose individual values are in the range of the numeric digits 0 through 9.

An alternative form of a declaration with initial values is illustrated with the variable definition in figure 6.5. This version does not use the reserved words ZERO, ZEROS, or ZEROES—a numeric digit is used instead. Note that *numeric literals* composed of digits differ from nonnumeric literals composed of alphanumerics, as shown in figure 6.6. As you can see, the nonnumeric literal is enclosed, or delimited, by quotes. The numeric literal is not.

Nonnumeric literals can contain any alphanumeric character that is a part of the computer's alphanumeric set (e.g., EBCDIC, ASCII). Every alphanumeric character set includes the blank. Were it not for the blank, then nonnumeric literals would not require delimiting quotes.

Note that REPORT-TITLE-1's literal in figure 6.6 is composed of eight leading blanks and the phrase SAMPLE OUTPUT. Delimiting quotes are required for nonnumeric literals so that the COBOL compiler can determine how many blanks prefaced the first nonblank character: in this case, the *S* in SAMPLE.

Numeric values, on the other hand, are composed of combinations of the single digit values 0 through 9. Numeric values cannot contain letters of the alphabet, special characters, or blanks. Because blanks cannot be a part of a numeric literal, quotes are not needed. In fact, if you attempt to include quotes, the COBOL compiler issues an appropriate error message and disregards the literal altogether.

Numeric variables can be initialized to any numeric value as long at that value is not smaller than the smallest value permitted, nor larger than the largest value permitted. The concept of negative numeric values as well as noninteger values will be discussed in a later chapter. In this chapter, all numeric values are assumed to be positive integers or zero.

Truncation

Despite their major differences, alphanumeric variables and numeric variables have much in common, and they are defined in similar ways. Compare the declarations depicted in figure 6.7. Each variable is preceded by a level number. The value 01 indicates that each variable is not to be subdivided into its constituent parts. Every variable name is followed by its attributes. Alphanumeric variables are symbolized by an X,

```
       01  CURRENT-LINE        PICTURE IS 9(03) VALUE 0.
```

Figure 6.5
Numeric literals composed of numeric digits

```
       01  CURRENT-LINE        PICTURE IS 9(03) VALUE 0.
       01  REPORT-TITLE-1      PICTURE IS X(80) VALUE
      -    "              SAMPLE OUTPUT".
```

Figure 6.6
Comparing numeric and nonnumeric literals

```
       01  BLANK-LINE          PICTURE IS X(80) VALUE SPACES.
       01  CURRENT-LINE        PICTURE IS 9(03) VALUE ZERO.
```

Figure 6.7
Similarities in data definitions

```
           ADD 1 TO CURRENT-LINE.
```

Figure 6.8
Overflow and truncation of results

and numeric variables are symbolized by a 9. The variable type (X or 9) is followed by a repetition factor. The variable BLANK-LINE contains 80 storage locations while CURRENT-LINE contains only 3. CURRENT-LINE has been initialized to zero while BLANK-LINE will contain blanks. Each declaration is followed, as are all COBOL statements, by a period.

As it is defined, CURRENT-LINE may contain numeric values ranging from 0 to a maximum of 999. For example, assume that during the execution of a COBOL program, CURRENT-LINE does contain the value 999. What effect will the executable statement in figure 6.8 have upon CURRENT-LINE? CURRENT-LINE would try to reset to 1,000; but because CURRENT-LINE can hold only 999 as its largest value, *truncation* would occur. Therefore, CURRENT-LINE would be reset to 0. *Truncation* is a serious concern. Unlike truncation of alphanumeric variables, which occurs on the right, truncation of numeric variables occurs on the left. The leading digit, or the 1 of 1,000, is truncated and the remaining VALUE of 000 is assigned to CURRENT-LINE.

Truncation
The loss of a character because a value larger than the field could hold was transferred to that field.

Programmers must realize the significance of truncation on reports. Otherwise, they may sometimes define the size of a variable as less than what it should be. An example is a numeric variable named NET-PAY in a payroll program. Here a variable smaller than needed might cause someone to receive a paycheck for an incorrect amount. The high-order digits would be lost because truncation occurs on the left in numeric fields. A paycheck that should be issued for \$550 would instead be issued for \$50.

Using Numeric Variables to Control Page Overflow

The last concept to be introduced in this chapter, while not complex, is the most difficult of those introduced. We need to modify the program logic model for report programs to include the ability to place page titles and trailers on each and every page of the report produced by a COBOL program. Our pseudo-coding in earlier versions was limited to printing report titles only at the very beginning of the printed output and trailers only at the end. Our present program cannot determine when it has reached the end of a page. The program operates as if the report is one long page with its size limited only by the number of input records to be transferred to the printed output.

Inquiry Statement
A statement that allows the programmer to determine the value a field contains. It is also known as a decision statement.

Our pseudo-coded report program, shown in figure 6.9, has been altered to include a new concept. That concept is known as an inquiry statement. The *inquiry statement* permits the designers of computer programs to determine if a particular event has occurred. In the case of the revised report program, the event being monitored is known as page overflow. Page overflow occurs when the bottom of the current page is reached and it is impossible to print another line without crossing over to the top of the next page. Printing a data line at the top of the next page would yield a rather unusual looking report because report titles are supposed to be the only printed matter at the very top of any report page. Also, the last item that should appear at the bottom of any page within the report is not a data line, but a report trailer. Thus, our report program must be modified to count printed lines as they are displayed. We can then anticipate the end of the page, print the report trailer at the bottom of the current page, and print the report titles at the top of the next page.

You might have noticed that what started as a very simple report program has rapidly become quite complex. We will review the report program's development at this time.

The program started as a simple device for reading input records from a file of records. Each record from the input file was to be printed on the printer. We then modified this program to include the ability to print report titles on the very first page produced, and report trailers on the very last page of the report.

The unusual appearance of reports produced by this report program was not desirable. The report should contain report titles and trailers on each and every page. As a result, we further modified the report program to include the pseudo-coding statements that monitor page overflow.

When the page overflow condition arises during the printing of our report, the program responds by (1) printing the report trailers at the bottom of the current page and (2) printing the report titles at the top of the next page. The program then continues to transfer input records from the input file to the printer using the ACCEPT and DISPLAY statements. This process continues until all of the records in the input file are transferred to the printer.

To translate the program logic represented in pseudo-code into a working COBOL program, you must know the COBOL equivalent of each pseudo-coded statement. You have probably noticed that these equivalents are fairly obvious. In fact, they have a direct, one-to-one correspondence. To enhance your ability to convert the monitoring statement, you need to know the equivalent of the inquiry statement. Once again the COBOL equivalent of the pseudo-coded statement is strikingly similar, as shown in figure 6.10.

Statement Format

```
B
↓
IF condition
  THEN
     statement(s).
```

```
Statement    Pseudo-coded statement
number

   1          print report titles
   2          read very first inventory record image.
   3          do the print and read process
                    until
                          record image is blank.
   4          print report trailers
   5          stop execution.

   6      print and read process
   7          if next line printed would overflow page
   8               then
   9                    print report trailer
  10                    print report titles
  11                    reset line counter
  12                    add 1 to page counter.
  13          print that very same record image.
  14          read another inventory record image
```

Figure 6.9
Report logic with page overflow inquiry

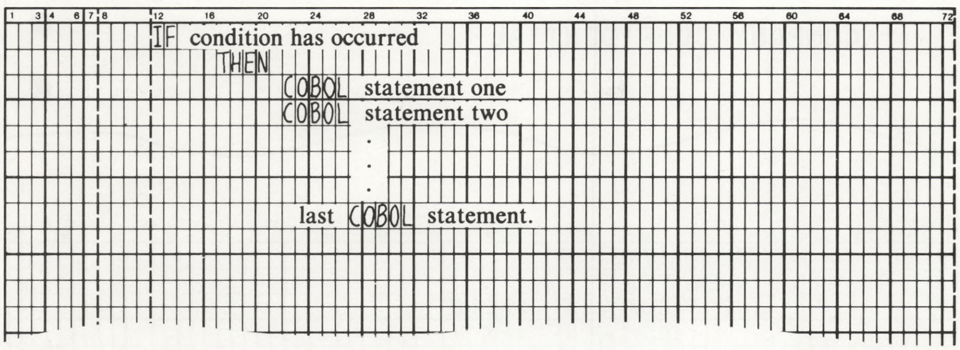


Figure 6.10
COBOL inquiry or IF statement

Terminating the IF Statement

Only one period is used in the entire COBOL IF statement. That period can only appear after the very last statement that is a part of the THEN portion of the IF statement. This is best illustrated by translating the inquiry statement in the pseudo-code to its COBOL IF counterpart, as shown in figure 6.10.

In chapter 5 we decided to produce a printed report that appears as shown in figure 6.11. The report contains seven titles and one trailer. In translating the inquiry statement from pseudo-code to COBOL, the equivalent of the inquiry statement, as seen in figure 6.12, is possible. Note that the only period occurs at the very end of the THEN portion of the IF statement. You should recall that the period marks the end of any COBOL statement. By placing the period after the last of the indented COBOL verbs, we have insured that if CURRENT-LINE is equal to 55, all of those DISPLAY instructions will be executed.

A period placed after each of the indented instructions drastically changes the actions of the computer. Note the somewhat different format of the COBOL IF statement shown in figure 6.13.

The use of *indentation* can be very misleading. The indented statements appearing after THEN imply that those statements will only be executed if CURRENT-LINE is equal to 55. However, by placing a period after the first indented statement, we have terminated the IF instruction prematurely. As a result, only the first DISPLAY will be executed if CURRENT-LINE is equal to 55. The remaining statements will be executed whether CURRENT-LINE is 55 or not.

Indentation
A programmer's device used to illustrate which program statements are subordinate to other statements.

Figure 6.11
Payroll report

```
            ACME MACHINE COMPANY
    LEISURE AND SPORTS PRODUCTS DIVISION
          MINNEAPOLIS SALES OFFICE

           WEEKLY PAYROLL REPORT

SOCIAL          EMPLOYEES       HR     HR
SECURITY        NAME            WK     WG
NUMBER

215-50-3145     WEINER, HD      18     300
197-42-1508     BAGGINS, FX     25     275
     .              .            .      .
     .              .            .      .
     .              .            .      .

          COMPANY CONFIDENTIAL
```

Figure 6.12
Use of period when terminating IF statement

```
           IF CURRENT-LINE = 55
               THEN
                   DISPLAY BLANK-LINE
                   DISPLAY TRAILER-1
                   DISPLAY BLANK-LINE
                   DISPLAY BLANK-LINE
                   DISPLAY TITLE-1
                   DISPLAY TITLE-2
                   DISPLAY TITLE-3
                   DISPLAY BLANK-LINE
                   DISPLAY TITLE-4
                   DISPLAY BLANK-LINE
                   DISPLAY BLANK-LINE
                   DISPLAY TITLE-5
                   DISPLAY TITLE-6
                   DISPLAY TITLE-7
                   DISPLAY BLANK-LINE
                   MOVE 11 TO CURRENT-LINE.
```

```
       IF CURRENT-LINE = 55
           THEN
               DISPLAY BLANK-LINE.
               DISPLAY TRAILER-1.
               DISPLAY BLANK-LINE.
               DISPLAY BLANK-LINE.
               DISPLAY TITLE-1.
                   .
                   .
                   .
```

Figure 6.13
Premature termination of IF statement

Page 1 of 2

```
       PRINT-AND-READ-PROCESS SECTION.
       P-A-R-P-001.
      *    ****  THIS SECTION REPRESENTS THE PRINT AND READ PUMP
      *          MENTIONED EARLIER.  WHEN INVOKED, IT PRINTS THE
      *          PREVIOUSLY READ CARD IMAGE AND THEN READS YET
      *          ANOTHER.
           IF CURRENT-LINE = 55
              THEN
                  DISPLAY BLANK-LINE
                  DISPLAY TRAILER-1
                  DISPLAY BLANK-LINE
                  DISPLAY BLANK-LINE
                  DISPLAY TITLE-1
                  DISPLAY TITLE-2
                  DISPLAY TITLE-3
                  DISPLAY BLANK-LINE
                  DISPLAY TITLE-4
                  DISPLAY BLANK-LINE
                  DISPLAY BLANK-LINE
                  DISPLAY TITLE-5
                  DISPLAY TITLE-6
                  DISPLAY TITLE-7
                  DISPLAY BLANK-LINE
                  MOVE 11 TO CURRENT-LINE.
```

Page 2 of 2

```
           ADD 1 TO CURRENT-LINE.
           DISPLAY RECORD-IMAGE.
           ACCEPT RECORD-IMAGE.
```

Figure 6.14
Page overflow and the IF statement

This last point is better illustrated by the entire print-and-read-process in figure 6.14. Here, if CURRENT-LINE is equal to 55, then control progresses to the THEN portion of the IF statement. That is, each of the DISPLAY instructions is executed in turn with the MOVE instruction being executed last. After the MOVE instruction, control proceeds first to the ADD instruction and then to the DISPLAY instruction that prints the contents of RECORD-IMAGE. Next, a new RECORD-IMAGE is read, and control leaves the section and returns to the invoking PERFORM, or iterative, instruction.

If CURRENT-LINE is not equal to 55, the computer is directed to skip all of the instructions associated with the THEN portion of the IF statement. In other words, because the IF condition is false, control goes immediately to the ADD instruction.

Figure 6.15
Misplaced period in IF statement

Page 1 of 2

```
       PRINT-AND-READ-PROCESS SECTION.
       P-A-R-P-001.
      *    **** THIS SECTION REPRESENTS THE PRINT AND READ PUMP
      *         MENTIONED EARLIER.  WHEN INVOKED, IT PRINTS THE
      *         PREVIOUSLY READ CARD IMAGE AND THEN READS YET
      *         ANOTHER.
           IF CURRENT-LINE = 55
               THEN
                   DISPLAY BLANK-LINE
                   DISPLAY TRAILER-1.
                   DISPLAY BLANK-LINE
                   DISPLAY BLANK-LINE
                   DISPLAY TITLE-1
                   DISPLAY TITLE-2
                   DISPLAY TITLE-3
                   DISPLAY BLANK-LINE
                   DISPLAY TITLE-4
                   DISPLAY BLANK-LINE
                   DISPLAY BLANK-LINE
                   DISPLAY TITLE-5
                   DISPLAY TITLE-6
                   DISPLAY TITLE-7
                   DISPLAY BLANK-LINE
                   MOVE 11 TO CURRENT-LINE.
```

Page 2 of 2

```
           ADD 1 TO CURRENT-LINE.
           DISPLAY RECORD-IMAGE.
           ACCEPT RECORD-IMAGE.
```

Note what happens in figure 6.15, which is another example of a misplaced period in an IF statement and illustrates a common error. A period has been inadvertently placed after the instruction that prints the trailer message. This period terminates the IF statement sooner than the programmer had anticipated. What is the result?

If CURRENT-LINE is not equal to 55, control passes from the IF condition to the statement immediately after the IF. That instruction is the first COBOL statement following the terminating period, or in this instance, the DISPLAY instruction that prints a blank line—supposedly after the trailer message. Consequently, the blank lines and the title messages will be printed every time the iterative statement invokes this section in this PROCEDURE DIVISION. So, we will have a set of titles produced for each RECORD-IMAGE that is read and subsequently printed. This is not what the programmer intended.

Use of Indentation

Also, note that in the IF statements shown thus far, many separate source statements are used to contain the entire inquiry instruction. This is in keeping with a programming style you are encouraged to follow. The alternative is to place the entire statement on as few lines as possible, as depicted in figure 6.16. This last version is undoubtedly easier to keypunch because it requires fewer source statements; however, it is more difficult to interpret because the THEN portion of the statement contains many executable statements. If the THEN portion of the IF statement contains more than one statement, as is usually the case, the stylistic approach shown in figure 6.14 is much more helpful to the reader.

```
       PRINT-AND-READ-PROCESS SECTION.
       P-A-R-P-001.
      *    ****  THIS SECTION REPRESENTS THE PRINT AND READ PUMP
                 MENTIONED EARLIER. WHEN INVOKED, IT PRINTS THE
                 PREVIOUSLY READ CARD IMAGE AND THEN READS YET
                 ANOTHER.
           IF CURRENT-LINE = 55 THEN DISPLAY BLANK-LINE
           DISPLAY TRAILER-1 DISPLAY BLANK-LINE DISPLAY
           BLANK-LINE DISPLAY TITLE-1 DISPLAY  TITLE-2
           DISPLAY TITLE-3 DISPLAY BLANK-LINE  DISPLAY
           TITLE-4 DISPLAY BLANK-LINE  DISPLAY  BLANK-LINE
           DISPLAY  TITLE-5 DISPLAY TITLE-6 DISPLAY TITLE-7
           DISPLAY  BLANK-LINE MOVE 9 TO CURRENT-LINE.    ADD 1
           TO CURRENT-LINE.   DISPLAY RECORD-IMAGE.   ACCEPT
           RECORD-IMAGE.
```

Figure 6.16
Lack of stylistic formatting

=	(or)	EQ	(or)	EQUAL
>		GT		GREATER THAN
<		LT		LESS THAN

You may also combine operators by using the NOT specification. For instance,

NOT =	(or)	NEQ	(or)	NOT EQUAL
NOT >		NOT GT		NOT GREATER THAN
NOT <		NOT LT		NOT LESS THAN

Figure 6.17
Relational operators

Relational Operators

The IF verb is followed immediately by an inquiry condition. That condition is usually composed of a variable defined in the WORKING-STORAGE SECTION, a literal, and a *relational operator*. A relational operator is the mechanism that tells the computer how CURRENT-LINE and 55 are to be compared. The valid relational operators are shown in figure 6.17. It should be pointed out that the IF condition can also be stated as

```
IF 55 = CURRENT-LINE
```

Relational Operator
The mechanism used in inquiry statements to specify how a variable and a value are to be related. For instance, is the variable equal to, less than, etc., a chosen literal value?

The following alternative is also possible

```
IF CURRENT-LINE = PAGE-SIZE
```

Here PAGE-SIZE is a variable defined in the WORKING-STORAGE SECTION with an initial value which is, in terms of the number of printed lines it can hold, the size of the printed page. So, variables can be compared to literals and to other variables.

Summary

Our original report program was incapable of printing report titles and trailer messages on each and every page of printed output. This problem was overcome by restructuring the pseudo-coded version to count which line the program was currently printing. When the end of the page was reached, trailer messages were printed, title messages were printed on the next page, and the line count was reset to count the lines of a new page.

Line counting required the introduction of a new variable type—the numeric variable. Numeric variables are the only variables that can be used for arithmetic operations because these variables can hold only numeric values. Alphanumeric variables can hold both numeric and nonnumeric characters and, thus, are not eligible for use in arithmetic operations.

Numeric variables are defined in the same manner as alphanumeric variables. Definitions occur in the WORKING-STORAGE SECTION. Each has a PICTURE IS clause and can have a VALUE clause. At compilation time, numeric variables are initialized by the use of numeric literals, which are not delimited by quotes. Furthermore, numeric variables are right justified and truncation drops the most significant (largest) digit. Great care should be taken to define variables large enough for their intended use, and to initialize them to their proper values.

Page overflow monitoring requires numeric variables for line counting, and also an inquiry instruction to determine when page overflow is about to occur. This capability is provided in COBOL with the IF statement.

The IF statement is a powerful facility. In IF statements, it is very important to properly indent subordinate instructions and to properly place terminating periods.

Our entire COBOL report program now appears as shown in figure 6.18.

Figure 6.18
Final report program

Page 1 of 5

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.   REPORT1.
           SKIP2
       ENVIRONMENT DIVISION.
           SKIP2
       DATA DIVISION.
       WORKING-STORAGE SECTION.
      *
      *    ****      ALL PROGRAM COUNTERS
      *
       01  CURRENT-LINE PICTURE IS 9(2) VALUE 0.
           SKIP2
      *
      *    ****      ALL REPORT TITLES AND TRAILERS
      *
       01  TITLE-1            PICTURE IS X(80) VALUE
           "                              ACME MACHINE COMPANY".
       01  TITLE-2            PICTURE IS X(80) VALUE
           "                       LEISURE AND SPORTS PRODUCTS DIVISION".
       01  TITLE-3            PICTURE IS X(80) VALUE
           "                            MINNEAPOLIS SALES OFFICE".
       01  TITLE-4            PICTURE IS X(80) VALUE
           "                             WEEKLY PAYROLL REPORT".
       01  TITLE-5            PICTURE IS X(80) VALUE
```

Page 2 of 5

```
           "SOCIAL           EMPLOYEES                   HR       HR
      -    "                   ".
       01  TITLE-6        PICTURE IS X(80) VALUE
           "SECURITY         NAME                        WK       WG
      -    "                   ".
       01  TITLE-7        PICTURE IS X(80) VALUE
           "NUMBER
      -    "                   ".
       01  TRAILER-1      PICTURE IS X(80) VALUE
           "                         COMPANY CONFIDENTIAL".
       01  BLANK-LINE     PICTURE IS X(80) VALUE SPACES.
           SKIP2
      *    ****ALL OTHER VARIABLES
      *
       01  RECORD-IMAGE           PICTURE IS X(80).
           SKIP2
       PROCEDURE DIVISION.
       MAIN-CONTROL SECTION.
       M-C-001.
      *    ****      PRINT THE REPORT TITLES THAT MUST APPEAR AT THE
      *              BEGINNING OF THE REPORT.
           DISPLAY TITLE-1.
           DISPLAY TITLE-2.
           DISPLAY TITLE-3.
```

Figure 6.18
(continued)

Page 3 of 5

```
           DISPLAY BLANK-LINE.
           DISPLAY TITLE-4.
           DISPLAY BLANK-LINE.
           DISPLAY BLANK-LINE.
           DISPLAY TITLE-5.
           DISPLAY TITLE-6.
           DISPLAY TITLE-7.
           DISPLAY BLANK-LINE.
           MOVE 11 TO CURRENT-LINE.
           SKIP2
       M-C-002.
      *    ****    THIS SECTION OF THE PROGRAM SERVES TO CONTROL THE
      *            PRINT AND READ PUMP SECTION THAT FOLLOWS.  TO
      *            START THE PUMP ONE CARD IMAGE IS READ AND THEN
      *            PUMP IS EXECUTED.  WHEN THE BLANK CARD IS FOUND,
      *            THE PUMP TERMINATES ALONG WITH THIS PROGRAM'S
      *            EXECUTION.
           ACCEPT RECORD-IMAGE.
           PERFORM PRINT-AND-READ-PROCESS
               UNTIL
                   RECORD-IMAGE = SPACES.
           DISPLAY BLANK-LINE.
```

Page 4 of 5

```
           DISPLAY TRAILER-1.
           STOP RUN.
           SKIP2
       PRINT-AND-READ-PROCESS SECTION.
       P-A-R-P-001.
      *    ****    THIS SECTION REPRESENTS THE PRINT AND READ PUMP
      *            MENTIONED EARLIER.  WHEN INVOKED, IT PRINTS THE
      *            PREVIOUSLY READ CARD IMAGE AND THEN READS YET
      *            ANOTHER.
           IF CURRENT-LINE = 55
               THEN
                   DISPLAY BLANK-LINE
                   DISPLAY TRAILER-1
                   DISPLAY BLANK-LINE
                   DISPLAY BLANK-LINE
                   DISPLAY TITLE-1
                   DISPLAY TITLE-2
                   DISPLAY TITLE-3
                   DISPLAY BLANK-LINE
                   DISPLAY TITLE-4
                   DISPLAY BLANK-LINE
                   DISPLAY BLANK-LINE
                   DISPLAY TITLE-5
                   DISPLAY TITLE-6
```

Page 5 of 5

```
                   DISPLAY TITLE-7
                   DISPLAY BLANK-LINE
                   MOVE 11 TO CURRENT-LINE.
           ADD 1 TO CURRENT-LINE.
           DISPLAY RECORD-IMAGE.
           ACCEPT RECORD-IMAGE.
```

Questions for Review

1. Why is it to the programmer's advantage to assign initial values to variables when they are defined, rather than using an execution instruction such as MOVE?
2. Why do nonnumeric literals require delimiting quotes while numeric literals do not?
3. Why is the impact of truncation likely to be much more severe for numeric variables than alphanumeric variables? What can be done to avoid truncation?
4. Did the pseudo-coded report program shown in prior chapters produce a multiple- or single-page output? Why?
5. The newly discussed multiple-page logic arbitrarily defines a page to be 55 lines long. If a trailer message is to be included among those 55 lines, should end-of-page occur at line 55? Should it occur earlier? Why?
6. What is the possible impact of misplacing the termination period in an IF statement? Placed too soon? Too late?
7. What constitutes the use of indentation when writing IF statements?

Problems

1. Construct several alternatives to the following variable declaration. (Hint: Reserved words that begin with *Z*.)

```
01  CURRENT-LINE   PICTURE IS 9(2) VALUE 0.
```

2. After examining the following variable definitions, construct a diagram that shows both the number of primary memory positions allocated to each variable and the value in each respective position.

```
01  A     PICTURE IS 9(12) VALUE 12389.
01  B     PICTURE IS 9(08) VALUE 246933.
01  C     PICTURE IS 9(04) VALUE 0.
```

3. In exercise 2, what would occur when the value in A is moved to C? Would the value in A be altered?
4. In exercise 2, what would occur when the value in B is moved to A?
5. Examine the page overflow segment extracted from a COBOL program. Would it function as it was intended?

```
IF CURRENT-LINE = 55
   THEN
      DISPLAY TITLE-01.
      DISPLAY TITLE-02.
      DISPLAY BLANK-LINE.
      MOVE ZERO TO CURRENT-LINE.
```

Programming Assignments

1. Shortt and Wilson Realtors specialize in high-rise condominium sales in newly converted (i.e., from rental property) buildings. The firm has developed a new property sale system featuring a Property Master File. Each unit that is available for sale is represented by a record within this file. Each record adheres to the following format.

Record Position	*Data Type*	*Variable*
01–04	N	Project identification code
05–11	N	Unit identification
12–19	N	Sale price ($)
20–49	AN	Unit's legal description
50–50	N	Number of bedrooms
51–51	N	Number of bathrooms
52–57	N	Square footage
58–80	—	Unused

The report to be produced can be found in figure 6.19.

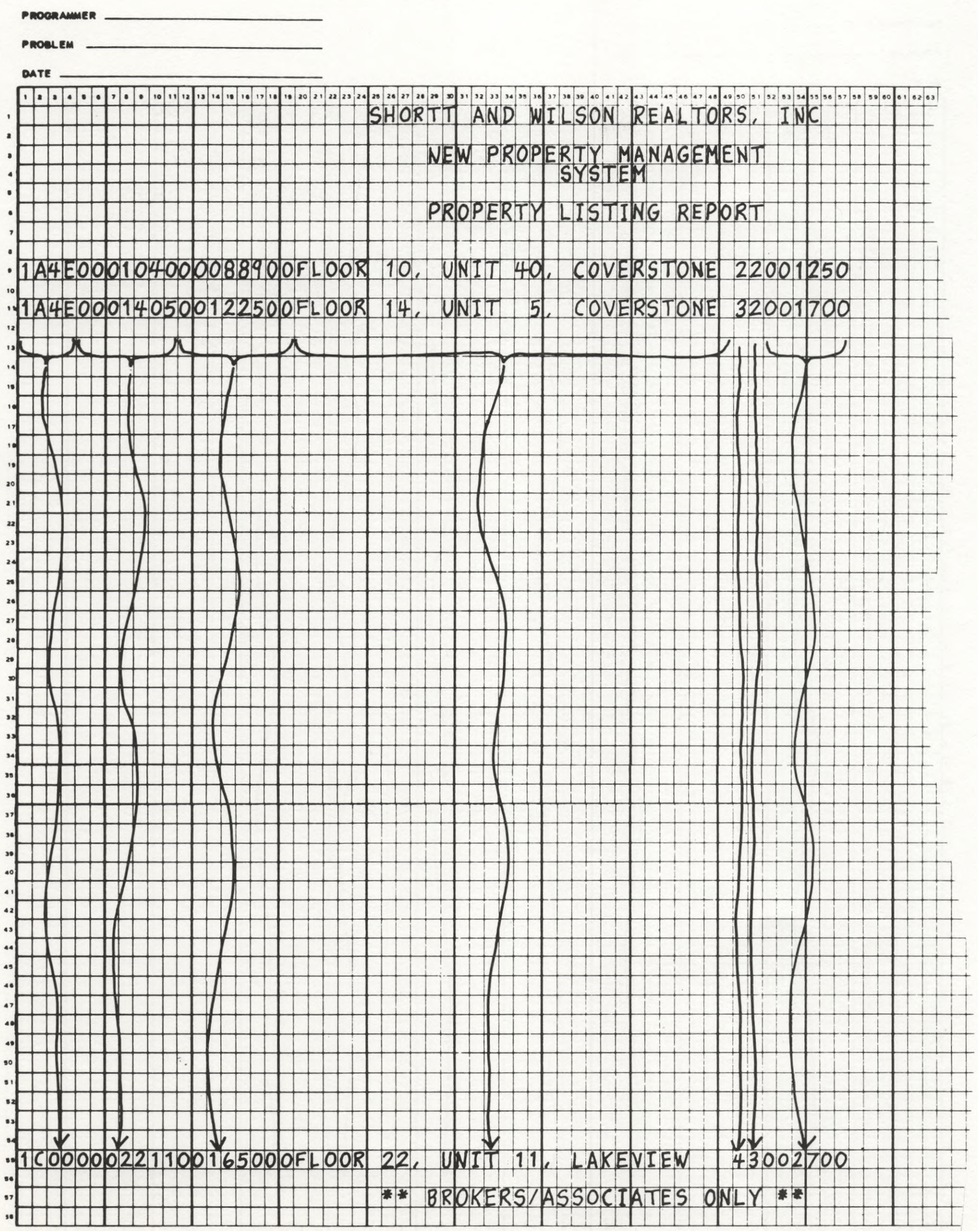


Figure 6.19
Shortt and Wilson realty report

2. New Wave Systems is a wholesale firm that markets used luxury automobiles in the southeastern U.S. Federal regulations require that regional used-car wholesalers report, on a monthly basis, all acquisitions and their status at the time of the report. New Wave Systems records all acquisition and subsequent resale data in its Used Car Master File. Each car is represented by a record within this file in the following manner.

Record Position	*Data Type*	*Variable*
01–20	AN	Auto identification number
21–24	N	Acquisition cost ($)
25–25	AN	Status of auto
26–30	N	Resale price ($)
31–40	AN	Auto make and model
41–42	N	Year of auto
43–52	AN	Dealer identification (i.e., resale)
53–80	—	Unused

The report to be produced can be found in figure 6.20.

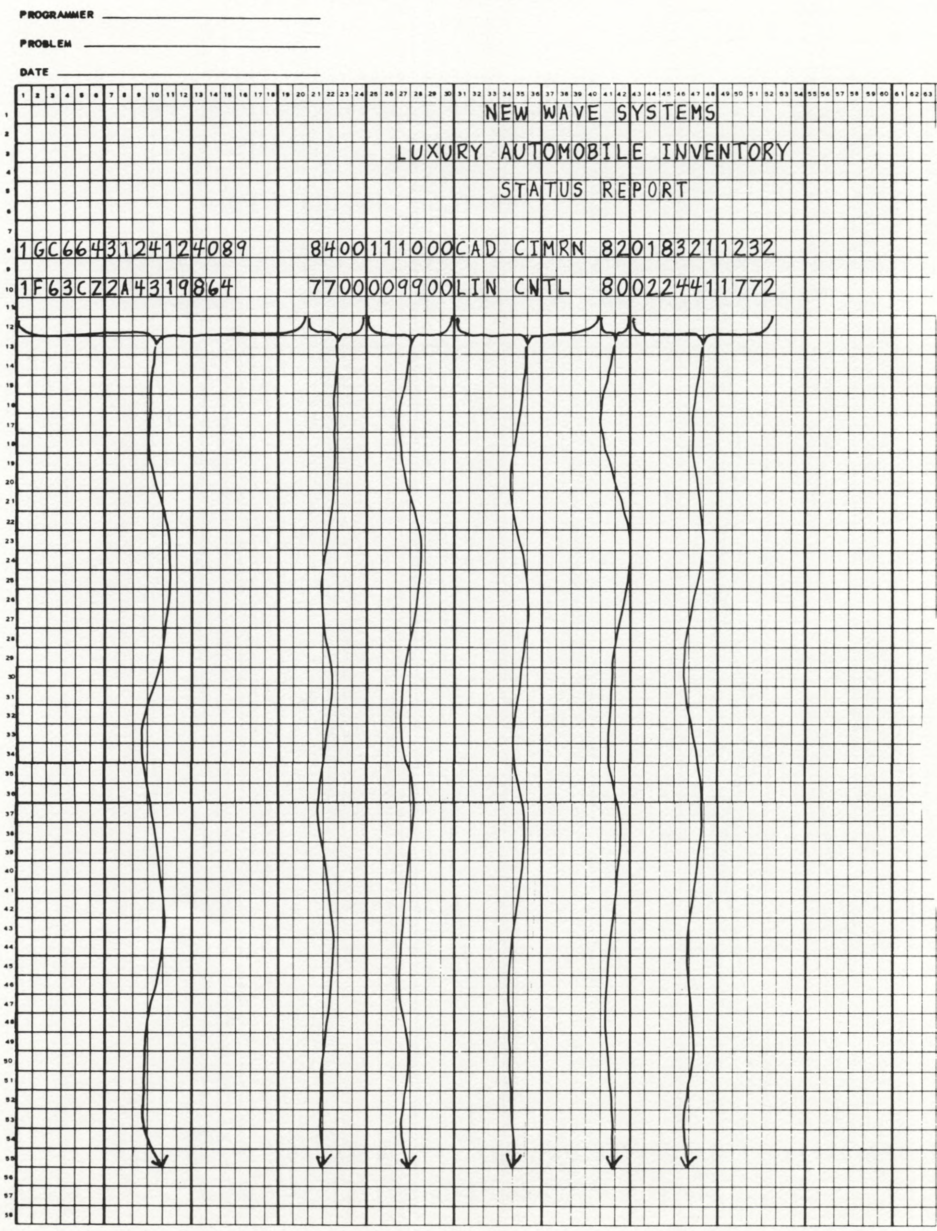


Figure 6.20
New Wave Systems auto status report

Report Titles and Record Structures

7

Chapter Outline

In chapter 6, the COBOL report program was modified to include the ability to sense a condition known as page overflow. When a page overflow condition occurs, all report trailers are printed, along with the title for the next page. None of these title or trailer messages have included a page number. However, real computer-produced reports usually contain page numbers in either the titles or the trailers located on each page in the report.

For the moment, assume that page numbers will be printed as part of the topmost title on each printed page. How is this accomplished? The first title used in one of the sample report programs is shown in figure 7.1.

The phrase ACME MACHINE COMPANY will be printed by a DISPLAY instruction so that it is centered within the 80-character print line. The title phrase will be preceded and followed by blanks. In short, the topmost report title will appear as shown in figure 7.2.

Now we would like the title to appear as shown in figure 7.3. In this case, we cannot define the desired report title as a single 01-level variable declaration. Why? When variable declarations were first introduced, it was noted that 01 in the A-Margin represents a record description to the COBOL compiler. While the record being defined as a 01-level item might contain numerous *fields,* the sole description is that of one large record with no field access whatsoever.

Field
A collection of consecutive record positions that constitute a single item of data.

Figure 7.1
Topmost report title

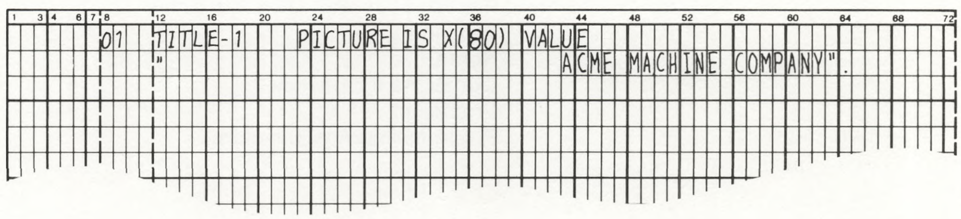


Figure 7.2
Appearance of top title in report

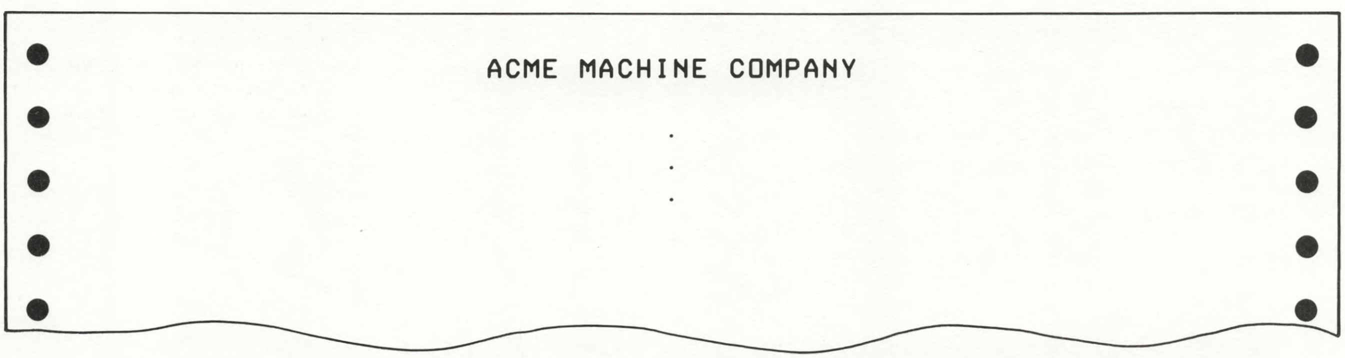
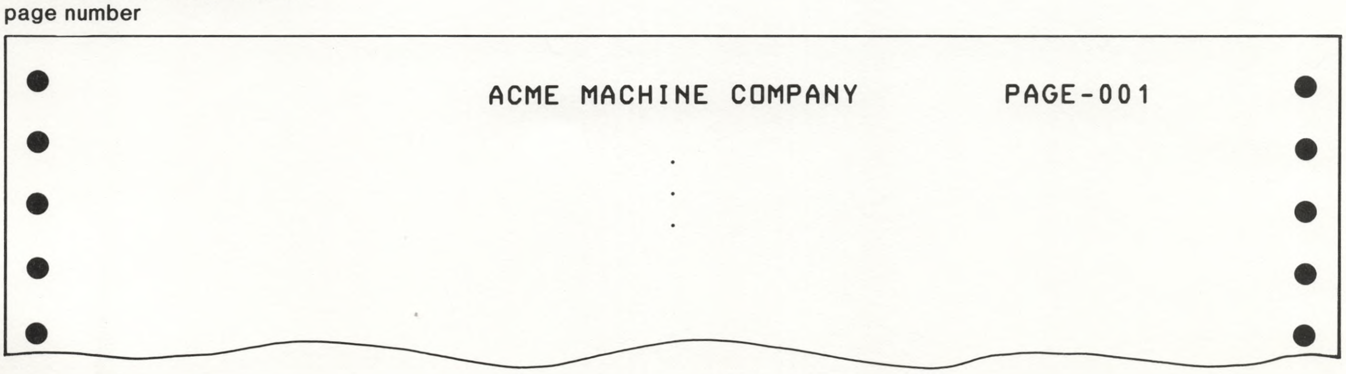


Figure 7.3
Preferred format featuring a page number



Our proposed revision to TITLE-1 requires *field access.* Figure 7.3 illustrates that the field to which access is desired is the three-digit page number located at the extreme right-hand side of the title. Each time a new page is printed, the current page number must be incremented by one and the report title printed reflecting the new page number. How would the incrementing instruction be written? Would ADD 1 TO TITLE-1 be acceptable? No, this instruction is incorrect.

Field Access
A manner of defining an entire record so that any specific field within that record can have its contents retrieved or altered without disrupting the remainder of the record.

Arithmetic instructions, such as the ADD instruction, can operate only on numeric variables. Numeric variables are defined as PICTURE IS 9. . . rather than as PICTURE IS X. . . . Numeric variables can contain only the digits 0 through 9 in each of their positions. Alphanumeric variables, on the other hand, can hold any numeric, alphabetic, blank, or special character. TITLE-1 has been defined as an alphanumeric variable. So, it is not possible to involve all of TITLE-1 in an arithmetic operation. The challenge, then, is to redefine our TITLE-1 variable so that the page number portion of that variable can be accessed. Page numbers are numeric and can be incremented by the ADD instruction.

Record Structures

Examine again the desired report title shown in figure 7.3. This particular report title consists of five distinct parts. They are:

1. Leading blanks—these blanks push the phrase ACME MACHINE COMPANY to the approximate center of the printed line.
2. Actual title—this is the phrase that identifies the name of the company for whom the report is being prepared—ACME MACHINE COMPANY.
3. Separating blanks—these are the blank positions in the title between the corporate name and the phrase PAGE-001.
4. Page phrase—this portion of the literal (PAGE-) identifies the three-digit numeric value that follows as a page number.
5. Page number.

A mechanism that defines this five-part report title and provides access to each of its five fields is called *record structure.* This field-by-field declaration provides the programmer with access to each of the five parts of the report title. The programmer may not need to access each individual field. Field access may actually be needed for only one of the many fields a record may contain. However, to access any individual field, the title must be structured field by field rather than as a single record.

Record Structure
A collection of field definitions whose sum total equals a complete record for input or output data.

Statement Format

```
A   B
↓   ↓
01  recordname.
    levelnumber variablename PICTURE IS attributes [VALUE
    literal].
        .
        .
        .
        .
        .
        .
    levelnumber variablename PICTURE IS attributes [VALUE
    literal].
```

Figure 7.4
Basic format of a record structure

```
       01  TITLE-1.
           02  __________          PICTURE IS X(27) VALUE SPACES.
           02  __________          PICTURE IS X(26) VALUE
           "ACME MANUFACTURING COMPANY".
           02  __________          PICTURE IS X(24) VALUE
           "                       PAGE-".
           02  __________          PICTURE IS 9(3) VALUE ZERO.
```

Figure 7.5
Completing record structure using FILLER

```
       01  TITLE-1.
           02  FILLER              PICTURE IS X(27) VALUE SPACES.
           02  FILLER              PICTURE IS X(26) VALUE
           "ACME MANUFACTURING COMPANY".
           02  FILLER              PICTURE IS X(24) VALUE
           "                       PAGE-".
           02  PAGE-NUMBER         PICTURE IS 9(3) VALUE ZERO.
```

The record structure version of the report title appears in figure 7.4. A large number of source statements are now required to declare a single report title. Remember, the record structure definition is required because we desire to access a field within the title (i.e., page number). If field access was not required, then the earlier declaration requiring only two source statements would be sufficient. But field access requires records to be structured field by field.

You have probably noticed several interesting characteristics of record structures. First, the 01 level no longer has a PICTURE IS clause immediately following the name of the record being defined. Instead, the variable name is followed by a period. This period indicates that the variable is further defined field by field. Second, the subordinate fields all begin with level 02 in the B-Margin. In general, higher-level numbers are inferior to smaller-level numbers. Thus, to define the fields in an 01 level, 02s must be used; to define the fields in a 02 level, 03s must be used; and so on. All level numbers greater than 01 must start in the B-Margin—a rule of the COBOL language. Third, each of the 02-level definitions looks exactly like a 01-level variable. That is, each field variable name is followed by a PICTURE IS clause and a VALUE clause, where appropriate. Each definition is terminated by a period.

Variable names have not yet been supplied for each of the 02 level definitions in figure 7.4. An explanation of special variable names is required here before they are added to the example.

The record structure is required before we can access any one of the fields in the report title. In order to increment the page number by 1 every time a new page is started, we need a numeric variable that can be manipulated by the ADD instruction. The four other fields in the structure do not need to be manipulated. The corporate name and the phrase *PAGE-* will never be altered. The blanks surrounding the company name must always be present. In situations where fields within structures will not be manipulated, COBOL provides the variable name to be used; that variable name is FILLER.

FILLER
A special field name that informs the COBOL language processor that access to the field is not desired. The field contains data whose reference or use is not desired.

Compare the record structures shown in figure 7.4 and figure 7.5. Only one of the 02 fields within the structure is given a name other than FILLER. That field is named PAGE-NUMBER. A programmer whose program contains this structure definition is able to manipulate only this PAGE-NUMBER field. The other fields cannot be manipulated because the COBOL compiler interprets the name FILLER as the name of a field that is not to be accessed. Record structures can have multiple field definitions named FILLER for all fields that will not be accessed.

Figure 7.6
Payroll report

```
                    ACME MACHINE COMPANY
            LEISURE AND SPORTS PRODUCTS DIVISION
                  MINNEAPOLIS SALES OFFICE

                      WEEKLY PAYROLL REPORT

SOCIAL            EMPLOYEES            HR     HR
SECURITY          NAME                 WK     WG
NUMBER

215-50-3145       WEINER, HD           18     310
197-42-1508       BAGGINS, FX          12     288

                      COMPANY CONFIDENTIAL
```

Figure 7.7
Time card record layout

Card columns	Type	Data
01–11	Alphanumeric	Social security number
12–19	N/A	Unused
20–32	Alphabetic	Employee's name
33–39	N/A	Unused
40–41	Numeric	Hours worked
42–49	N/A	Unused
50–53	Numeric	Hourly wage
54–80	N/A	Unused

Input and Output Record Structures

The COBOL report program described at the end of chapter 6 was designed to print a payroll report. That report, when produced by the program, is illustrated in figure 7.6. Do you notice anything unusual about the appearance of the titles, trailers, and data lines? The report titles seem to be off center, or the time card images are improperly positioned. We know that the title messages are centered because in the previous chapters we correctly centered them. The input record images, on the other hand, are produced in an 80–80 listing manner. In other words, the data on the time cards are printed exactly as they are shown. To center the data on the report, they must be centered on the time card.

To refresh your memory, figure 7.7 is the file and record layout of the time card file. It should be obvious that the fields within this record are not spread evenly in it. The Social Security number, for instance, starts in the first column of the record. Hourly wage, on the other hand, ends well before the end of the record. In fact, columns 54 through 80 are unused. It is not unusual for data to be spread unevenly on the input record.

A New Concept of a Print Line

Up to now we have assumed that the length of a printed line is exactly the same as the length of the input record. This is possible, but printed lines often correspond to the capacity of the line printer. Seldom is the size of the print line the same as that of the input record.

Many line printers can print more than 120 characters in each line of print. Indeed, some can print more than 144 characters. Few, however, print less than 120 characters. It is not unusual, therefore, to find COBOL report programs that assume the size of a printed line containing at least 120 characters.

The advantage of a larger print line may not be obvious at first. Input records are usually defined to hold as many fields as they can accommodate. That is, an input record seldom contains significant numbers of fields dedicated to unused space. Consequently, the time card just described is quite unusual because it holds fifty unused positions.

When records are packed with usable fields, on the other hand, it is common to find those fields physically adjacent to one another without any unused fields to serve as *buffers.* A more realistic version of our time card record layout is represented in figure 7.8. Note here that only one field is actually unused. The other fields contain important data, and they are positioned one right after the other. What happens if this input record is printed by a DISPLAY instruction? Assuming that the input record is printed exactly as it is described above, it appears as an unbroken stream of seventy-nine characters. Such a report is very difficult to read and interpret.

Buffer
In this context only, the use of blanks to separate nonblank fields as a means of improving the reader's ability to comprehend the data contained in a computer-produced report.

Because a printed equivalent of the input record would be difficult to read (i.e., everything would appear without intervening blanks), we need to reformat this input record by using a larger print line. A 120-character print line provides the opportunity (1) to place each of the input fields in a balanced print line and (2) to use blank spaces that will enable you to easily read and understand the report.

Examine both of the record definitions found in figure 7.9. Would the MOVE instruction accomplish the desired result? If an input record is difficult to read, would it be easier to read if it were simply placed in a larger record? No. It is true that the sequence of the two instructions will cause the image of the time card record, an eighty-character variable, to be copied into a much larger variable for printing. But what will the DISPLAY instruction actually print?

Figure 7.8
Packed record structures

Card columns	Type	Data
01–11	Alphanumeric	Social security number
12–36	Alphanumeric	Employee's name
37–56	Alphanumeric	Employee's street address
57–66	Alphanumeric	Employee's city
67–68	Alphanumeric	Employee's state
69–73	Numeric	Employee's zip code
74–75	Numeric	Hours worked
76–79	Numeric	Hourly wage
80–80	N/A	Unused

Figure 7.9
Unsuccessful attempt to reformat input records for printing

```
       01  TIME-CARD-INPUT       PICTURE IS X(80).

       01  TIME-CARD-OUTPUT      PICTURE IS X(120).

           MOVE TIME-CARD-INPUT TO TIME-CARD-OUTPUT.
           DISPLAY TIME-CARD-OUTPUT.
```

The MOVE instruction copies the contents of one variable into another variable. In the example just cited, the contents of TIME-CARD-INPUT will be copied into the variable named TIME-CARD-OUTPUT. TIME-CARD-OUTPUT is defined as being much larger than TIME-CARD-INPUT. What happens when a smaller variable's contents are copied into a larger variable? The answer corresponds to what happens when a variable is assigned a literal with the VALUE clause.

When the contents of smaller variables are moved into larger variables, the move is left justified for alphanumeric variables and right justified for numeric variables. Both TIME-CARD-INPUT and TIME-CARD-OUTPUT are alphanumeric variables. As a result, the contents of TIME-CARD-INPUT will be left justified within TIME-CARD-OUTPUT. The forty characters located at the extreme right of TIME-CARD-OUTPUT will be automatically set to blanks.

This means that all fields that comprise TIME-CARD-INPUT will be printed in TIME-CARD-OUTPUT exactly as they were arranged in the input record. All the move to a larger field accomplishes is to place forty blanks after the fields from the input record. The unintelligible string of characters remains. The challenge is to allocate those forty blank characters so that they separate the fields within the input record.

Record Structure Definition of the Output Record

By defining TIME-CARD-OUTPUT as a record structure with field access rather than as a single variable, we can force the forty blank positions to be spread throughout the record. This can be done so that each of the fields to be printed will be surrounded by blanks. This makes it easier for the reader to determine where one field ends and the next begins.

A possible record structure is shown in figure 7.10. As you can see, each of the fields has been buffered with several blanks that isolate each field for ease of reading. What had been seventy-nine characters, one after the other, is now divided into groups separated by blanks. For example, the Social Security number no longer runs into employee's name, etc.

TIME-CARD-OUTPUT is defined as a structure with field-by-field access possible, if we elected to use it. Is it possible to treat a structure both as a structure and as a single variable? The answer is yes. To treat TIME-CARD-OUTPUT as a single,

```
       01  TIME-CARD-INPUT         PICTURE IS X(80).

       01  TIME-CARD-OUTPUT.
           02 FILLER               PICTURE IS X(07) VALUE SPACES.
           02 SOCIAL-SECURITY      PICTURE IS X(11).
           02 FILLER               PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-NAME        PICTURE IS X(25).
           02 FILLER               PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-STREET      PICTURE IS X(20).
           02 FILLER               PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-CITY        PICTURE IS X(10).
           02 FILLER               PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-STATE       PICTURE IS X(02).
           02 FILLER               PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-ZIPCODE     PICTURE IS 9(05).
           02 FILLER               PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-HOURS       PICTURE IS 9(02).
           02 FILLER               PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-WAGE        PICTURE IS 9(04).
           02 FILLER               PICTURE IS X(06) VALUE SPACES.

           MOVE TIME-CARD-INPUT TO TIME-CARD-OUTPUT.
           DISPLAY TIME-CARD-OUTPUT.
```

Figure 7.10
Unsuccessful attempt to utilize field access

large field rather than a structure made up of smaller fields, use the MOVE statement shown in figure 7.10. The reason is simple: The MOVE instruction in figure 7.10 refers to both the sending and receiving areas by their 01 record names. So, even if the records are defined field by field, these subordinate definitions will be ignored and a group item transfer will occur. The 80-character group known as TIME-CARD-INPUT will be copied into the 120-character group known as TIME-CARD-OUTPUT.

What happens if the variable definition of TIME-CARD-INPUT is unchanged and the two instructions in figure 7.10 are executed? TIME-CARD-INPUT is still accessed as a single record and field-by-field access for TIME-CARD-INPUT is not possible. Consequently, the MOVE instruction will transfer the eighty characters in TIME-CARD-INPUT into the first eighty positions of TIME-CARD-OUTPUT. The remaining forty positions will be set to blanks.

To restructure the printed result as we desire, we must transfer the contents of TIME-CARD-INPUT to TIME-CARD-OUTPUT field by field. To do this, TIME-CARD-INPUT must be defined as a structure. In other words, the sending, or source, record must also be defined as a record structure to provide field access.

A New Concept of the Input Record

To have field-by-field access to the data contained in TIME-CARD-INPUT, we must redefine this particular record as a record composed of many fields, as shown in figure 7.11. Can we now access each of the fields in the input record and transfer them to the print line as the illustration suggests? Not quite! Every time you refer to a structure by its name, you are treating the structure as if it were defined as a single-field record—not a record that permits individual field access. In order to transfer each of the fields within TIME-CARD-INPUT to TIME-CARD-OUTPUT, we must copy them from one structure to the other field by field. As a result, the sequence of instructions shown in figure 7.12 is required. This example may be somewhat confusing.

Qualification of Nonunique Variable Names

When the rules for choosing variable names were first explained, it was pointed out that the COBOL compiler requires each variable to be given a unique name. At that time all of our variables were defined as 01-level items and the concept of the record structure had not been introduced.

The rules have not changed. All 01-level variables must have unique names. When the 01-level variables are further defined by 02-, 03-, etc., level variables in a structure, then those inferior, or subordinate, fields need not be unique.

Notice how each of the two structures discussed in this chapter use the same variable names. Both TIME-CARD-INPUT and TIME-CARD-OUTPUT contain the same data. The only difference between them is that the input record is not suitably structured for printing while the output record is. In this instance, nonunique variable names can be chosen, meaning that the same variable name can be used in both structures. The same variable name, however, cannot be used twice within the same structure.[1] We must now examine the MOVE instructions of the form

```
MOVE SOCIAL-SECURITY TO SOCIAL-SECURITY.
```

If particular variable names have been used twice, each instruction can have two different interpretations. Either you are asking the COBOL compiler to

```
MOVE SOCIAL-SECURITY OF TIME-CARD-INPUT TO
     SOCIAL-SECURITY OF TIME-CARD-OUTPUT.
```

or

```
MOVE SOCIAL-SECURITY OF TIME-CARD-OUTPUT TO
     SOCIAL-SECURITY OF TIME-CARD INPUT.
```

Page 1 of 2

```
       01  TIME-CARD-INPUT.
           02 SOCIAL-SECURITY       PICTURE IS X(11).
           02 EMPLOYEE-NAME         PICTURE IS X(25).
           02 EMPLOYEE-STREET       PICTURE IS X(20).
           02 EMPLOYEE-CITY         PICTURE IS X(10).
           02 EMPLOYEE-STATE        PICTURE IS X(02).
           02 EMPLOYEE-ZIPCODE      PICTURE IS 9(05).
           02 EMPLOYEE-HOURS        PICTURE IS 9(02).
           02 EMPLOYEE-WAGE         PICTURE IS 9(04).
           02 FILLER                PICTURE IS X(01).

       01  TIME-CARD-OUTPUT.
           02 FILLER                PICTURE IS X(07) VALUE SPACES.
           02 SOCIAL-SECURITY       PICTURE IS X(11).
           02 FILLER                PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-NAME         PICTURE IS X(25).
           02 FILLER                PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-STREET       PICTURE IS X(20).
           02 FILLER                PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-CITY         PICTURE IS X(10).
           02 FILLER                PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-STATE        PICTURE IS X(02).
           02 FILLER                PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-ZIPCODE      PICTURE IS 9(05).
```

Page 2 of 2

```
           02 FILLER                PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-HOURS        PICTURE IS 9(02).
           02 FILLER                PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-WAGE         PICTURE IS 9(04).
           02 FILLER                PICTURE IS X(06) VALUE SPACES.

           MOVE TIME-CARD-INPUT TO TIME-CARD-OUTPUT.
           DISPLAY TIME-CARD-OUTPUT.
```

Figure 7.11
Unsuccessful attempt to reformat with input record structure

```
           MOVE SOCIAL-SECURITY TO SOCIAL-SECURITY.
           MOVE EMPLOYEE-NAME TO EMPLOYEE-NAME.
                    .
                    .
                    .
           MOVE EMPLOYEE-WAGE TO EMPLOYEE-WAGE.
           DISPLAY TIME-CARD-OUTPUT.
```

Figure 7.12
Attempt to transfer fields from structure to structure

Figure 7.13
Field-by-field transfer

```
MOVE SOCIAL-SECURITY OF TIME-CARD-INPUT TO
     SOCIAL-SECURITY OF TIME-CARD-OUTPUT.
MOVE EMPLOYEE-NAME OF TIME-CARD-INPUT TO
     EMPLOYEE-NAME OF TIME-CARD-OUTPUT.
     .
     .
     .
MOVE EMPLOYEE-WAGE OF TIME-CARD-INPUT TO
     EMPLOYEE-WAGE OF TIME-CARD-OUTPUT.
DISPLAY TIME-CARD-OUTPUT.
```

Qualification
The means through which variables whose names are not unique are identified by owner record structure.

Of the two alternatives, the first is likely to produce the more desired outcome. The phrase OF TIME-CARD-INPUT (IN TIME-CARD-INPUT could also be used) is known as a *qualification.* Variable name qualification must be used when confusion might arise as to which of the variables (with the same name) you wish to manipulate. The instructions to reformat the input record and DISPLAY the results now look like that shown in figure 7.13. However, providing so many source statements just to reformat the input record is inefficient; and, for this reason, the designers of COBOL also created a multiple MOVE instruction.

Statement Format

```
B
↓
MOVE variablename [OF recordname] TO
     variablename [OF recordname].
```

Multiple MOVE Instruction

Earlier in this chapter it was pointed out that if both TIME-CARD-INPUT and TIME-CARD-OUTPUT were defined as structures, the COBOL instructions in figures 7.9, 7.10, and 7.11 would not give the desired results. Obviously, the programmer wanted to reformat the input record so that the printed counterpart was easier to read. After all, that was why both variables were altered to be structures rather than single fields at the 01 level.

Unfortunately, because the MOVE instruction referred to the variables using their 01 level, or major group names, the transfer of data from the sending variable to the receiving variable did not take place field by field as desired. Instead, the eighty characters in the first variable were copied into the second variable in one operation—as a group. This required that our growing COBOL program be further expanded to include a series of qualified MOVE instructions, with each field to be copied requiring its own separate transfer operation. A more desirable alternative is shown in figure 7.14. The CORRESPONDING option of the MOVE instruction is a rather fascinating one. In fact, this particular option was designed for just the set of circumstances covered in this chapter. It was designed specifically to reformat input records to produce a more appealing printed result.

Statement Format

```
B
↓
MOVE CORRESPONDING recordname TO recordname.
```

The CORRESPONDING option of the MOVE instruction has more applications than just reformatting input records. This option transfers like-named fields from one record structure to another. The fields do not have to appear in the same order in either structure. The matching process is carried out automatically by the COBOL compiler on the basis of matching variable names. In other words, the COBOL compiler generates the actual computer-executable instructions that transfer the contents of each field in the sending structure to the CORRESPONDING field in the receiving structure.

"MOVE CORRESPONDING TIME-CARD-INPUT TO
TIME-CARD-OUTPUT."

Fields that correspond

1. SOCIAL-SECURITY
2. EMPLOYEE-NAME
3. EMPLOYEE-STREET
4. EMPLOYEE-CITY
5. EMPLOYEE-STATE
6. EMPLOYEE-ZIPCODE
7. EMPLOYEE-HOURS
8. EMPLOYEE-WAGE

Figure 7.14
Fields that CORRESPOND—TIME-CARD-INPUT and TIME-CARD-OUTPUT

Figure 7.15
Sample report with date and time

```
                              ACME MACHINE COMPANY                             PAGE-001
                                                                               05-02-82
                       LEISURE AND SPORTS PRODUCTS DIVISION                    13:04:22
                            MINNEAPOLIS SALES OFFICE
                              WEEKLY PAYROLL REPORT

SOCIAL        EMPLOYEES     EMPLOYEES            EMPLOYEES  EMP   EMP     HRS  HRLY
SECURITY      NAME          STREET               CITY       STE   ZPCDE   WKD  WAGE
NUMBER                      ADDRESS              ADDRESS    CDE

                              COMPANY CONFIDENTIAL
```

A note of caution about the use of the CORRESPONDING option is required here. Frequently, beginning programmers attempt to MOVE the contents of a record for which field access has not been developed to another record that does have field access. In such an instance, the sending record has no fields. It is a long stream of characters. CORRESPONDING assumes that:

1. field access exists for both sending and receiving records, and
2. records have fields identically named.

If nothing corresponds, then nothing is MOVEd.

Also, FILLER fields are not eligible for transfer in the CORRESPONDING option of the MOVE. After all, if the name FILLER is chosen, obviously the field is of little importance.

Report Titles with Date and Time

It is quite common to include the date that a report was generated as part of a report title like the one shown in figure 7.15. Where would such a date originate?

Prior to the comprehensive operating systems that are common today, report programs were developed to acquire the date either from an input file or through a dialogue with the computer operator. Either alternative provided opportunities for error. The wrong date could be placed on a record destined for the input file, or the operator might inadvertently supply an incorrect date.

Computer Clock
A hardware clock that maintains both the current time of day as well as the day of the year. These hardware-controlled values are accessible only through a request (i.e., ACCEPT) issued to the computer's operating system software.

Today most operating systems provide access to the *computer clock* mechanism so that an application program can copy the present date into a record structure in its DATA DIVISION. Note the example in figure 7.16. The ACCEPT statement is put to a new, additional use. Not only can the ACCEPT verb be the means by which input record images are read, but the verb can also supply the current data from the computer's clock.

The date acquired in figure 7.16 is six digits in length and contains data in the form YYMMDD. Field access to each of the date's major components has been established so that, for purposes of this particular report, the date value can be reformatted to contain separating hyphens and a different sequence (i.e., MM-DD-YY).

Statement Format

```
B
↓                            ⎧DATE⎫
ACCEPT recordname FROM ⎨DAY  ⎬
                             ⎩TIME⎭
```

Figure 7.16 also illustrates that the time of day can be acquired by the COBOL program using the ACCEPT statement. In this option of the ACCEPT verb, an eight-digit time of day is available: HHMMSSTH. The clock is a twenty-four-hour clock. Few report applications require a value for time so accurate that it is printed containing seconds, tenths of seconds, and hundredths of seconds. Consequently, the example in figure 7.16 ignores the two digits located at the extreme right, by naming them FILLER. However, such precision is available if desired.

Summary

All computer-produced reports must include page numbers. These page numbers can appear as part of the titles at the top of each page or, where applicable, the trailers at the bottom of each page. Title and trailer variables have, prior to this chapter, been defined solely through the use of 01-level declarations. In 01-level declarations, the entire record is described as one large alphanumeric field—often eighty characters in length. Page numbering requires the use of an incrementing (ADD) instruction and a numeric field upon which to operate. This need to access only the numeric portion of the title or trailer that contains the page number requires the introduction of the record structure.

The record structure provides an alternative to the earlier method of variable definition. Here a record is defined field by field. Fields that contain data are defined and accounted for, as well as those fields that serve to mark unused portions (FILLER). The record structure requires larger numbers of source statements. These extra source statements, however, must be used to gain access to a particular field within the record, while leaving the remaining fields unaltered.

The record definition that is described as a structure starts as all record definitions have started. The 01 level, however, is terminated by a period following the variable's structure name. All subordinate fields are assigned inferior (i.e., numerically larger) level numbers and begin in the B-Margin. Like all variable definitions, these subordinate declarations contain PICTURE IS, and, optionally, VALUE clauses. These field declarations are terminated by periods, too.

Also, our detail lines (i.e., any lines other than a title or trailer) have not been centered in our earlier examples, as were the title and trailer messages. By redefining the input record as a structure to provide field-by-field access, these data items can be transferred for printing to a print-line record that is also defined as a structure. A

Figure 7.16
Report titles with date and time

Augmented report titles — Page 1 of 3

```
       01  TITLE-1.
           02 FILLER                    PICTURE IS X(50) VALUE SPACES.
           02 FILLER                    PICTURE IS X(62) VALUE.
           "ACME MACHINE COMPANY".
           02 FILLER                    PICTURE IS X(05) VALUE.
           "PAGE-".
           02 PAGE-NUMBER               PICTURE IS 9(03) VALUE 1.
           SKIP1
       01  TITLE-1A.
           02 FILLER                    PICTURE IS X(112) VALUE SPACES.
           02 REPORT-DATE.
              03 TODAYS-MONTH           PICTURE IS 9(02).
              03 FILLER                 PICTURE IS X(01) VALUE "-".
              03 TODAYS-DAY             PICTURE IS 9(02).
              03 FILLER                 PICTURE IS X(01) VALUE "-".
              03 TODAYS-YEAR            PICTURE IS 9(02).
           SKIP1
       01  TITLE-1B.
           02 FILLER                    PICTURE IS X(112) VALUE SPACES.
           02 REPORT-TIME.
              03 CURRENT-HOUR           PICTURE IS 9(02).
              03 FILLER                 PICTURE IS X(01) VALUE ":".
              03 CURRENT-MINUTE         PICTURE IS 9(02).
              03 FILLER                 PICTURE IS X(01) VALUE ":".
```

New date and time record structures — Page 2 of 3

```
              03 CURRENT-SECOND         PICTURE IS 9(02).

       01  CLOCK-DATE.
           02 TODAYS-YEAR               PICTURE IS 9(02).
           02 TODAYS-MONTH              PICTURE IS 9(02).
           02 TODAYS-DAY                PICTURE IS 9(02).
           SKIP1
       01  CLOCK-TIME.
           02 CURRENT-HOUR              PICTURE IS 9(02).
           02 CURRENT-MINUTE            PICTURE IS 9(02).
           02 CURRENT-SECOND            PICTURE IS 9(02).
           02 FILLER                    PICTURE IS 9(02).
```

Revised PROCEDURE DIVISION instructions — Page 3 of 3

```
       M-C-001.
      *
      *    ****    PRINT THE REPORT TITLES THAT MUST APPEAR AT THE
      *            BEGINNING OF THE REPORT.
           DISPLAY BLANK-LINE.
           DISPLAY BLANK-LINE.
           DISPLAY TITLE-1.
           ACCEPT CLOCK-DATE FROM DATE.
           MOVE CORRESPONDING CLOCK-DATE TO
                              REPORT-DATE.
           DISPLAY TITLE-1A.
           ACCEPT CLOCK-TIME FROM TIME.
           MOVE CORRESPONDING CLOCK-TIME TO
                              REPORT-TIME.
           DISPLAY TITLE-1B.
           DISPLAY TITLE-2.
           DISPLAY TITLE-3.
                .
                .
```

separate print-line record can provide the intervening blanks needed to separate the end of one field from the beginning of the next. This is necessary in situations in which the input record is filled with data fields and very little unused space.

The reformatting process becomes more involved when two variables are used in the reading and printing of input records. In our example, both structures use the same variable names for their subordinate fields. In such circumstances, MOVE instructions to accomplish the reformatting operation require a qualifying clause. This qualifier identifies in which structure a nonunique variable belongs.

Figure 7.17
Current COBOL report program

Page 1 of 9

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.    RPT0701.
           SKIP2
       ENVIRONMENT DIVISION.
           SKIP2
       DATA DIVISION.
       WORKING-STORAGE SECTION.
      *
      *    ****    ALL PROGRAM COUNTERS
      *
       01  CURRENT-LINE                PICTURE IS 9(2) VALUE ZERO.
           SKIP2
      *
      *    ****    ALL REPORT TITLES AND TRAILERS
      *
       01  TITLE-1.
           02 FILLER                   PICTURE IS X(50) VALUE SPACES.
           02 FILLER                   PICTURE IS X(62) VALUE
           "ACME MACHINE COMPANY".
           02 FILLER                   PICTURE IS X(05) VALUE
           "PAGE-".
           02 PAGE-NUMBER              PICTURE IS 9(03) VALUE 1.
           SKIP1
       01  TITLE-1A.
```

Page 2 of 9

```
           02 FILLER                   PICTURE IS X(112) VALUE SPACES.
           02 REPORT-DATE.
              03 TODAYS-MONTH          PICTURE IS 9(02).
              03 FILLER                PICTURE IS X(01) VALUE "-".
              03 TODAYS-DAY            PICTURE IS 9(02).
              03 FILLER                PICTURE IS X(01) VALUE "-".
              03 TODAYS-YEAR           PICTURE IS 9(02).
           SKIP1
       01  TITLE-1B.
           02 FILLER                   PICTURE IS X(112) VALUE SPACES.
           02 REPORT-TIME.
              03 CURRENT-HOUR          PICTURE IS 9(02).
              03 FILLER                PICTURE IS X(01) VALUE ":".
              03 CURRENT-MINUTE        PICTURE IS 9(02).
              03 FILLER                PICTURE IS X(01) VALUE ":".
              03 CURRENT-SECOND        PICTURE IS 9(02).
           SKIP1
       01  TITLE-2                     PICTURE IS X(120) VALUE
           "                                        LEISURE AND SPORTS
      -    " PRODUCTS DIVISION".
           SKIP1
       01  TITLE-                      PICTURE IS X(120) VALUE
           "                                                MINNEAPOLIS
      -    " SALES OFFICE".
```

Variable qualification is not without its problems. When the record to be reformatted contains any significant number of subordinate fields, the programmer must write one—and possibly two—source statements for each field to be copied. A more pleasing alternative is the CORRESPONDING option of the MOVE instruction. The CORRESPONDING option transfers the contents of the fields in one structure to their identically named counterparts in another structure.

Our COBOL report program now appears as shown in figure 7.17. The program in figure 7.17, when executed, produces the report that is shown in figure 7.18.

Page 3 of 9

```
           SKIP1
       01  TITLE-4                 PICTURE IS X(120) VALUE
           "                                                  WEEKLY PAY
      -    "ROLL REPORT".
           SKIP1
       01  TITLE-5                 PICTURE IS X(120) VALUE
           "       SOCIAL         EMPLOYEES                     EMPLOYEE
      -    "S              EMPLOYEES    EMP    EMP     HRS     HRLY".
           SKIP1
       01  TITLE-6                 PICTURE IS X(120) VALUE
           "       SECURITY       NAME                          STREET
      -    "               CITY         STE    ZPCDE   WKD     WAGE".
           SKIP1
       01  TITLE-7                 PICTURE IS X(120) VALUE
           "       NUMBER                                       ADDRESS
      -    "               ADDRESS      CDE".
           SKIP1
       01  TRAILER-1               PICTURE IS X(120) VALUE
           "                                                  COMPANY CO
      -    "NFIDENTIAL".
           SKIP1
       01  BLANK-LINE              PICTURE IS X(120) VALUE SPACES
           SKIP2
      *    **** ALL OTHER VARIABLES
```

Page 4 of 9

```
      *
       01  CLOCK-DATE.
           02 TODAYS-YEAR          PICTURE IS 9(02).
           02 TODAYS-MONTH         PICTURE IS 9(02).
           02 TODAYS-DAY           PICTURE IS 9(02).
           SKIP1
       01  CLOCK-TIME.
           02 CURRENT-HOUR         PICTURE IS 9(02).
           02 CURRENT-MINUTE       PICTURE IS 9(02).
           02 CURRENT-SECOND       PICTURE IS 9(02).
           02 FILLER               PICTURE IS 9(02).
           SKIP1
       01  TIME-CARD-INPUT.
           02 SOCIAL-SECURITY      PICTURE IS X(11).
           02 EMPLOYEE-NAME        PICTURE IS X(25).
           02 EMPLOYEE-STREET      PICTURE IS X(20).
           02 EMPLOYEE-CITY        PICTURE IS X(10).
           02 EMPLOYEE-STATE       PICTURE IS X(02).
           02 EMPLOYEE-ZIPCODE     PICTURE IS 9(05).
           02 EMPLOYEE-HOURS       PICTURE IS 9(02).
           02 EMPLOYEE-WAGE        PICTURE IS 9(04).
           02 FILLER               PICTURE IS X(01).
           SKIP1
       01  TIME-CARD-OUTPUT.
```

Figure 7.17
(continued)


```
           02 FILLER                  PICTURE IS X(07) VALUE SPACES.
           02 SOCIAL-SECURITY         PICTURE IS X(11).
           02 FILLER                  PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-NAME           PICTURE IS X(25).
           02 FILLER                  PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-STREET         PICTURE IS X(20).
           02 FILLER                  PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-CITY           PICTURE IS X(10).
           02 FILLER                  PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-STATE          PICTURE IS X(02).
           02 FILLER                  PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-ZIPCODE        PICTURE IS 9(05).
           02 FILLER                  PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-HOURS          PICTURE IS 9(02).
           02 FILLER                  PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-WAGE           PICTURE IS 9(04).
           02 FILLER                  PICTURE IS X(06) VALUE SPACES.
           SKIP3
       PROCEDURE DIVISION.
           SKIP1
       MAIN-CONTROL SECTION.
       M-C-001.
      *
      *    ****     PRINT THE REPORT TITLES THAT MUST APPEAR AT THE
```



```
      *             BEGINNING OF THE REPORT.
           DISPLAY BLANK-LINE.
           DISPLAY BLANK-LINE.
           DISPLAY TITLE-1.
           ACCEPT CLOCK-DATE FROM DATE.
           MOVE CORRESPONDING CLOCK-DATE TO
                                   REPORT-DATE.
           DISPLAY TITLE-1A.
           ACCEPT CLOCK-TIME FROM TIME.
           MOVE CORRESPONDING CLOCK-TIME TO
                                   REPORT-TIME.
           DISPLAY TITLE-1B.
           DISPLAY TITLE-2.
           DISPLAY TITLE-3.
           DISPLAY BLANK-LINE.
           DISPLAY TITLE-4.
           DISPLAY BLANK-LINE.
           DISPLAY BLANK-LINE.
           DISPLAY TITLE-5.
           DISPLAY TITLE-6.
           DISPLAY TITLE-7.
           DISPLAY BLANK-LINE.
           MOVE 13 TO CURRENT-LINE.
           SKIP2
```

Page 7 of 9

```
      M-C-002.
      *
      *    ****    THIS SEGMENT OF THE PROGRAM SERVES TO CONTROL THE
      *            PRINT AND READ PUMP SECTION THAT FOLLOWS.  TO
      *            START THE PUMP ONE CARD IMAGE IS READ AND THEN
      *            PUMP IS EXECUTED.  WHEN THE BLANK CARD IS FOUND,
      *            THE PUMP TERMINATES.
           ACCEPT TIME-CARD-INPUT.
           PERFORM PRINT-AND-READ-PROCESS
               UNTIL
                   TIME-CARD-INPUT = SPACES.
           SKIP2
       M-C-003.
      *
      *    ****    NOW THAT THE PRINTING/READING PROCESS IS OVER,
      *            COMPLETE THE LAST PAGE OF THE REPORT BY PRINTING
      *            THE FINAL SET OF TRAILERS.
           DISPLAY BLANK-LINE.
           DISPLAY TRAILER-1.
           STOP RUN.
           SKIP2
       PRINT-AND-READ-PROCESS SECTION.
       P-A-R-P-001.
      *
```

Page 8 of 9

```
      *    ****    THIS SEGMENT CHECKS TO SEE IF PAGE OVERFLOW
      *            WILL OCCUR WITH THE PRINTING OF ONE ADDITIONAL
      *            LINE.  IF SO, THEN TRAILERS AND TITLES ARE
      *            PRINTED WHERE APPROPRIATE.
           IF CURRENT-LINE = 55
               THEN
                   DISPLAY BLANK-LINE
                   DISPLAY TRAILER-1
                   DISPLAY BLANK-LINE
                   DISPLAY BLANK-LINE
                   ADD 1 TO PAGE-NUMBER
                   DISPLAY TITLE-1
                   DISPLAY TITLE-1A
                   DISPLAY TITLE-1B
                   DISPLAY TITLE-2
                   DISPLAY TITLE-3
                   DISPLAY BLANK-LINE
                   DISPLAY TITLE-4
                   DISPLAY BLANK-LINE
                   DISPLAY BLANK-LINE
                   DISPLAY TITLE-5
                   DISPLAY TITLE-6
                   DISPLAY TITLE-7
                   DISPLAY BLANK-LINE
```

Page 9 of 9

```
                   MOVE 13 TO CURRENT-LINE.
           SKIP2
       P-A-R-P-002.
      *
      *    ****    THE LAST RECORD IMAGE READ IS REFORMATTED
      *            AND PRINTED.  THE NEXT RECORD IMAGE TO BE
      *            PROCESSED IS READ.
           ADD 1 TO CURRENT-LINE.
           MOVE CORRESPONDING TIME-CARD-INPUT TO
                              TIME-CARD-OUTPUT.
           DISPLAY TIME-CARD-OUTPUT.
           ACCEPT TIME-CARD-INPUT.
```

Figure 7.18
Final report with reformatted output

```
                    ACME MACHINE COMPANY                          PAGE-001
            LEISURE AND SPORTS PRODUCTS DIVISION                  05-02-82
                  MINNEAPOLIS SALES OFFICE                        13:04:22

                   WEEKLY PAYROLL REPORT

SOCIAL       EMPLOYEES      EMPLOYEES           EMPLOYEES   EMP  EMP    HRS  HRLY
SECURITY     NAME           STREET              CITY        STE  ZPCDE  WKD  WAGE
NUMBER                      ADDRESS             ADDRESS     CDE

215-50-3145  WEINER, HD     12007 BOBWHITE DR   CATHARPIN   VA   22018  40   0350
212-88-7865  ADENAY, TG     7604 FTN BLEAU DR   ALEXANDRIA  VA   22180  40   0375
034-62-8711  BARTFELD, CI   6007 KIRBY RD       BETHESDA    MD   20852  38   0575
077-22-8134  BRYERE, EJ     6707 SULKY LA       WOODBRIDGE  VA   22191  35   0425
532-01-2389  CHANG, I-LOK   10012 DICKENS AVE   BETHESDA    MD   20853  40   0750
654-88-6528  BURREL, TL     326 LINCOLN AVE     TAKOMA PK   MD   20912  33   0633
221-55-6732  BURNS, JL      511 UNDERWOOD ST    WASHINGTON  DC   20007  25   0350
434-77-8790  CAGLE, EM      9318 CAROLINE AVE   DAMASCUS    MD   20814  38   0525
882-28-1231  CHIANG, GX     17607 KIRK LA       BETHESDA    MD   20814  20   0325

                   COMPANY CONFIDENTIAL
```

Notes

1. It is possible to use the same variable name more than once in a given structure. However, this requires the use of structures within structures. Carefully examine the following example:

   An inventory system produces a report that notes the dates on which an item was first introduced to the inventory, was last replenished from the supplier, was last ordered by a customer, and so on. Specifically, many dates are being recorded within the inventory structure. A possible record structure for such a system follows:

```
01   INVENTORY-RECORD.
     02 ACQUISITION-INFORMATION.
        03 RECORDED-DATE               PICTURE IS X(06).
        03 BIN-LOCATION                PICTURE IS X....
              .
              .
              .
     02 SUPPLIER-INFORMATION.
        03 SUPPLIER-IDENTIFICATION     PICTURE IS ....
        03 RECORDED-DATE               PICTURE IS ....
              .
              .
              .
     02 SALES-INFORMATION.
        03 QUANTITY-SOLD-TODATE        PICTURE IS ....
        03 RECORDED-DATE               PICTURE IS ....
              .
              .
              .
              .
```

In this example, if we wanted to transfer the RECORDED-DATE from this input record to an output record, the following type of nonunique variable qualification would be required.

```
MOVE RECORDED-DATE OF SUPPLIER-INFORMATION TO ....
```

The only level of qualification required is that which uniquely identifies the variable. That is, the following would be unnecessary.

```
MOVE RECORDED-DATE OF SUPPLIER-INFORMATION
                   OF INVENTORY-RECORD TO ....
```

Two levels of qualification would be required only if there were two nonunique variables—RECORDED-DATE and SUPPLIER-INFORMATION.

Questions for Review

1. If PAGE-NUMBER is initialized to zero, what will be the number of the first page printed? So that the page number will be assigned page 1, where would you place an ADD instruction?
2. Define each of the remaining titles (TITLE-2 through TITLE-7) as record structures.
3. Would the MOVE instruction in figure 7.9 cause the input record to be reformatted for printing?
4. Would the MOVE instruction in figure 7.10 cause the input record to be reformatted for printing?
5. Would the MOVE instruction in figure 7.11 cause the input record to be reformatted for printing?

Problems

1. Carefully examine the following record structure. What, if anything, is wrong with this record structure that is to be printed using a DISPLAY instruction?

```
..1.........2.........3.........4.........5.........
       01  TIME-CARD-INPUT
           02  FILLER            PICTURE IS X(70) VALUE
           "                 "
           02  FILLER            PICTURE IS X(10) VALUE
           "ACME MANUFACTURING COMPANY".
           02 FILLER             PICTURE IS X(60) SPACES.
           02 FILER              PICTURE IS X(04) VALUE
           "PAGE-".
           02 FILER              PICTURE IS 9(03) VALUE "000."
```

2. Examine the following segments from a COBOL program. What, if anything, is wrong?

```
01  RECORD-A.
    02 A            PICTURE IS X.
    02 B            PICTURE IS X.
  .
  .
01  RECORD-B.
    02 C            PICTURE IS X.
    02 D            PICTURE IS X.
  .
  .
  .
    MOVE CORRESPONDING RECORD-A TO RECORD-B.
```

3. Examine the three following title lines. Define all three as record structures, using at least three subordinate fields. However, the last title line is to be defined so that field access is provided for a three-character abbreviation for the month of the year, a two-digit day, and a two-digit decade or year (e.g., 81).

```
HENDERSON SALES, INC.
INVENTORY STATUS REPORT
 AS OF JUN 12, 1981
```

Programming Assignments

1. Quatraplex Systems, a software development firm, is employed by the U.S. Army to design logistical support systems. One such system is dedicated to the record-keeping function required to monitor the armored transport vehicle parts inventory. The Army maintains three parts depots for armored transports and must maintain adequate levels of all parts if vehicle downtime is to be minimized.

   One report program in this system produces an itemized listing of vehicle parts that have been ordered from manufacturers but have not yet been received at the appropriate parts depot. The format of that report can be found in figure 7.19. The input file from which this report was produced contains a single eighty-character record for each back-ordered part. The format of each record is as follows:

Record Position	*Data Type*	*Variable*
01–15	AN	Vehicle part number
16–23	AN	Vendor identification number
24–35	AN	Repair order requiring the back-ordered part
36–41	N	Date ordered from vendor (YYMMDD)
42–47	N	Date promised by vendor (YYMMDD)
48–80	—	Unused

   The last two columns of the report in figure 7.19 are to be computed using the date from the computer's clock.

2. Columbia Concrete Products of Columbia, South Carolina, recently received a contract award to construct ten bridges along the Intercoastal Waterway on behalf of the U.S. Army Corps of Engineers. The contract, however, stipulates that Columbia must provide the corps with a log of all concrete deliveries for each contract at the contract's completion. These delivery logs will aid the U.S. Army Corps of Engineers in authenticating Columbia's request for final payment.

   At the present time, Columbia's data processing system is designed to produce a file that contains an eighty-position record that represents each delivery of concrete. These records identify the days, times, quantity of concrete delivered, and the projects to whom such deliveries were made.

   Unfortunately, Columbia does not possess a computer (report) program that will list only those deliveries made on behalf of the U.S. Army Corps of Engineers as shown in figure 7.20. Columbia's format of each record in the input file is as follows:

Record Position	*Data Type*	*Variable*
01–06	N	Delivery invoice number
07–12	N	Date of delivery
13–16	N	Time of delivery
17–18	N	Cubic yards delivered
19–20	AN	Project to whom delivered (= "08" if to Corps)
21–60	AN	Delivery site
61–80	AN	Name of driver

Figure 7.19
Quatraplex Systems report for U.S. Army

PROGRAMMER ______________

PROBLEM ______________

DATE ______________

REPORT LAYOUT FORM

PAGE ______ OF ______

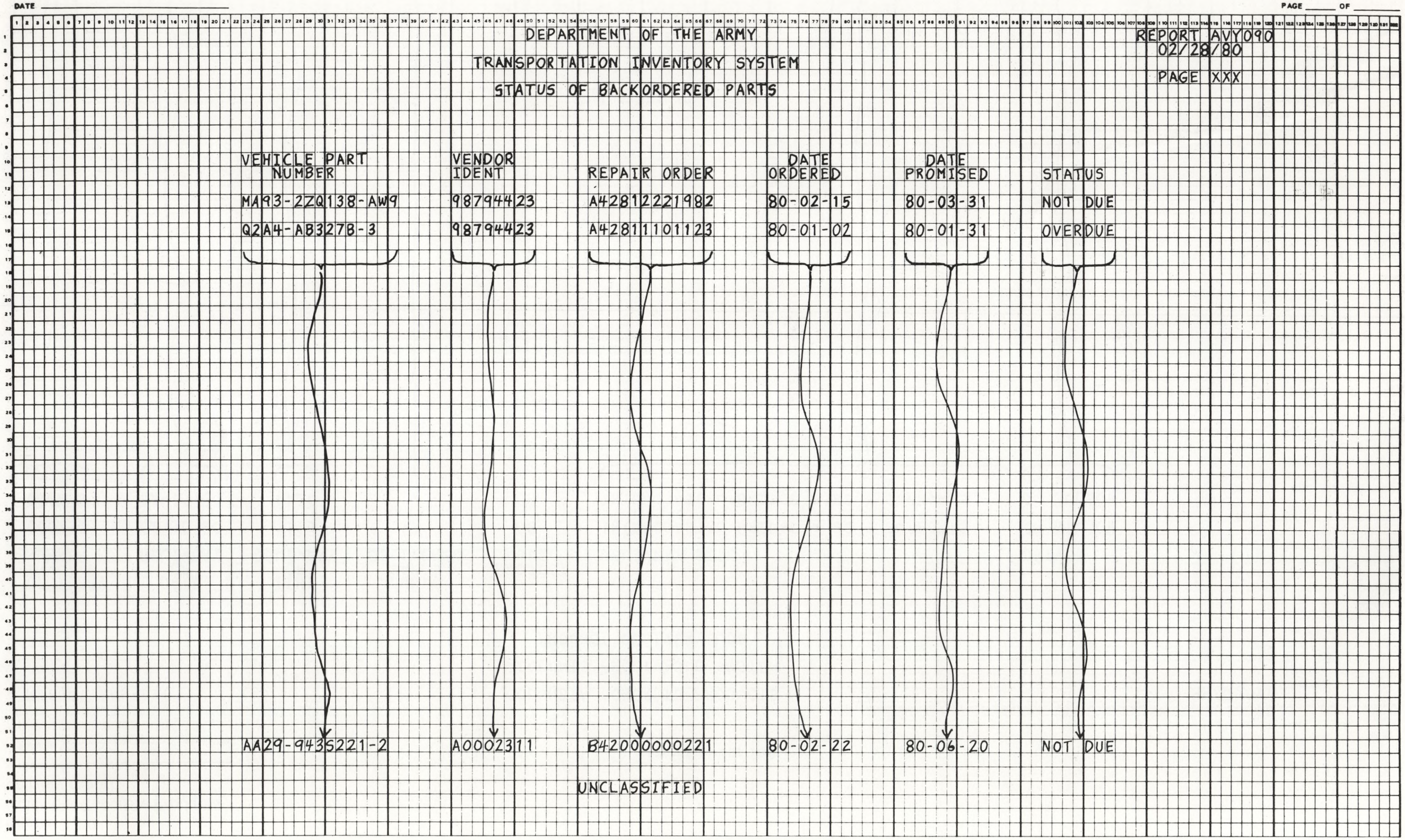
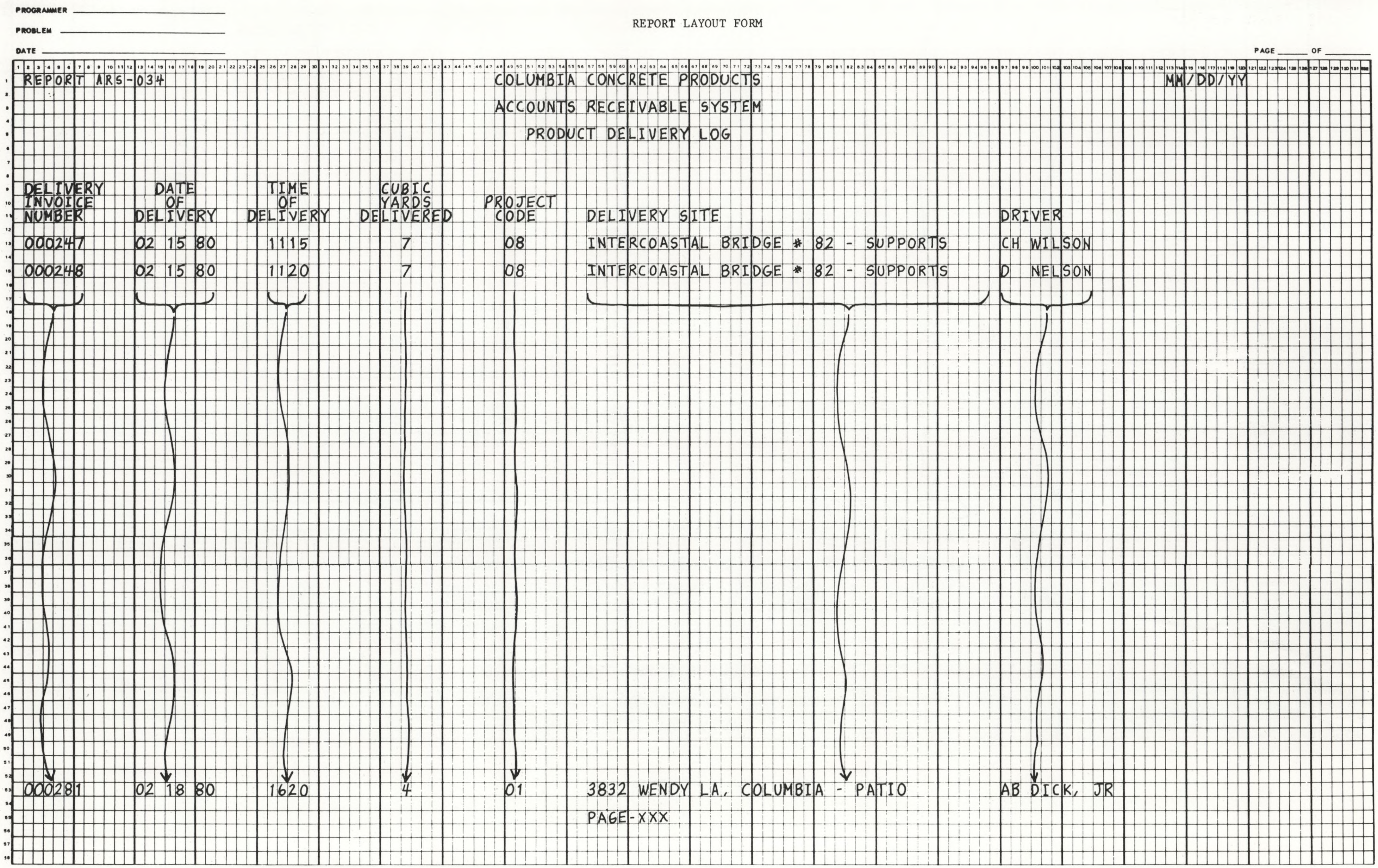


Figure 7.20
Columbia Concrete Products
delivery report



PROGRAMMER ____________

PROBLEM ____________

DATE ____________

REPORT LAYOUT FORM

PAGE ____ OF ____

REPORT ARS-034 COLUMBIA CONCRETE PRODUCTS MM/DD/YY

ACCOUNTS RECEIVABLE SYSTEM

PRODUCT DELIVERY LOG

DELIVERY INVOICE NUMBER	DATE OF DELIVERY	TIME OF DELIVERY	CUBIC YARDS DELIVERED	PROJECT CODE	DELIVERY SITE	DRIVER
000247	02 15 80	1115	7	08	INTERCOASTAL BRIDGE # 82 - SUPPORTS	CH WILSON
000248	02 15 80	1120	7	08	INTERCOASTAL BRIDGE # 82 - SUPPORTS	D NELSON
000281	02 18 80	1620	4	01	3832 WENDY LA, COLUMBIA - PATIO	AB DICK, JR

PAGE-XXX

Internal Subroutines

8

Chapter Outline

Figure 8.1
Report program logic

Statement number	Pseudo-coded statement

```
 1              print report titles
 2              read very first inventory record image.
 3              do the print and read process
                      until
                            record image is blank.
 4              print report trailers
 5              stop execution.

 6      print   and read process
 7              if next line printed would overflow page
 8                   then
 9                        print report trailer
10                        print report titles
11                        reset line counter
12                        increment page counter by 1.
13              increment line counter by 1
14              print that very same record image.
15              read another inventory record image
```

COBOL is a very wordy language. You may have the impression that even simple tasks, such as our report programs, require an excessive number of COBOL source statements. Comparatively speaking, these same reports may be easier to produce using other programming languages such as FORTRAN or PL/1. To be quite honest, however, we have been encouraging you to develop COBOL programs that actually use more statements than are required. Why? Because certain functions in our report program must be executed (i.e., repeated) at several different points. Report titles are an excellent example. Figure 8.1 represents the pseudo-coded logic that has evolved thus far. Note that report titles are printed in statements 1 and 10. Since, as the pseudocode suggests, this particular function occurs twice, two sets of COBOL statements are needed to print report titles. Duplication of large sections of COBOL statements leads to excessively large, cumbersome COBOL programs, which are difficult to comprehend.

Subroutine
A collection of COBOL statements that, when required, are executed by a PERFORM instruction. These permit the construction of shorter programs by eliminating duplicate segments of coding.

This chapter introduces a technique that can be used to reduce the size of COBOL programs. Basically, functions that appear two or more times are candidates for *subroutines.* A subroutine is a collection of COBOL source statements constituting a specific program function—such as the printing of report titles. A subroutine is segregated from the remainder of the COBOL program in a named section of the PROCEDURE DIVISION. In every place where the coding for the printing of report titles had been, there is now a single COBOL statement that causes the function to be PERFORMed.

Common Functions

Figure 8.2 shows the PROCEDURE DIVISION of the most-recent version of our COBOL report program. Notice two particular points in this portion of the program. The first is the paragraph known as M–C–001. The other is the paragraph known as P–A–R–P–001. At both points the program logic is designed to print titles at the top of each page of output. Each time this program prints the report titles, a total of fifteen COBOL instructions must be executed. Blank lines must be printed, followed by a page number incrementing operation, and so on. Because this operation is performed twice in our program, we have thirty statements (15 $\times$ 2) dedicated to this single function.

Figure 8.2 also illustrates the printing of report trailers. This particular function is also performed in two locations in the COBOL program. The first occurrence follows the iterative instruction that controls the print-and-read process. These instructions must be located here so that the trailer will not appear on the last page of program-produced output.

Figure 8.2
PROCEDURE DIVISION of report program

Page 1 of 4

```
       PROCEDURE DIVISION.
           SKIP1
       MAIN-CONTROL SECTION.
       M-C-001.
      *
      *    ****    PRINT THE REPORT TITLES THAT MUST APPEAR AT THE
      *            BEGINNING OF THE REPORT.

           DISPLAY BLANK-LINE.
           DISPLAY BLANK-LINE.
           ADD 1 TO PAGE-NUMBER.
           DISPLAY TITLE-1.
           DISPLAY TITLE-2.
           DISPLAY TITLE-3.
           DISPLAY BLANK-LINE.
           DISPLAY TITLE-4.
           DISPLAY BLANK-LINE.
           DISPLAY BLANK-LINE.
           DISPLAY TITLE-5.
           DISPLAY TITLE-6.
           DISPLAY TITLE-7.
           DISPLAY BLANK-LINE.
           MOVE 11 TO CURRENT-LINE.
```

Page 2 of 4

```
           SKIP2
       M-C-002.
      *
      *    ****    THIS SEGMENT OF THE PROGRAM SERVES TO CONTROL THE
      *            PRINT AND READ PUMP SECTION THAT FOLLOWS.  TO
      *            START THE PUMP ONE CARD IMAGE IS READ AND THEN
      *            PUMP IS EXECUTED.  WHEN THE BLANK CARD IS FOUND,
      *            THE PUMP TERMINATES.
           ACCEPT TIME-CARD-INPUT.
           PERFORM PRINT-AND-READ-PROCESS
               UNTIL
                   TIME-CARD-INPUT = SPACES.
           SKIP2
       M-C-003.
      *
      *    ****    NOW THAT THE PRINTING/READING PROCESS IS OVER,
      *            COMPLETE THE LAST PAGE OF THE REPORT BY PRINTING
      *            THE FINAL SET OF TRAILERS.
           DISPLAY BLANK-LINE.
           DISPLAY TRAILER-1.
           STOP RUN.
           SKIP2
       PRINT-AND-READ-PROCESS SECTION.
       P-A-R-P-001.
```

Figure 8.2
(continued)

Page 3 of 4

```
      *
      *    ****  THIS SEGMENT CHECKS TO SEE IF PAGE OVERFLOW
      *          WILL OCCUR WITH THE PRINTING OF ONE ADDITIONAL
      *          LINE.  IF SO, THEN TRAILERS AND TITLES ARE
      *          PRINTED WHERE APPROPRIATE.
           IF CURRENT-LINE = 55
               THEN
                   DISPLAY BLANK-LINE
                   DISPLAY TRAILER-1

                   DISPLAY BLANK-LINE
                   DISPLAY BLANK-LINE
                   ADD 1 TO PAGE-NUMBER
                   DISPLAY TITLE-1
                   DISPLAY TITLE-2
                   DISPLAY TITLE-3
                   DISPLAY BLANK-LINE
                   DISPLAY TITLE-4
                   DISPLAY BLANK-LINE
                   DISPLAY BLANK-LINE
                   DISPLAY TITLE-5
                   DISPLAY TITLE-6
                   DISPLAY TITLE-7
                   DISPLAY BLANK-LINE
```

Page 4 of 4

```
                   MOVE 11 TO CURRENT-LINE.

           SKIP2
       P-A-R-P-002.
      *
      *    ****  THE LAST RECORD IMAGE READ IS REFORMATTED
      *          AND PRINTED.  THE NEXT RECORD IMAGE TO BE
      *          PROCESSED IS READ.
           ADD 1 TO CURRENT-LINE.
           MOVE CORRESPONDING TIME-CARD-INPUT TO
                              TIME-CARD-OUTPUT.
           DISPLAY TIME-CARD-OUTPUT.
           ACCEPT TIME-CARD-INPUT.
```

The second occurrence of the statements dedicated to printing report trailers appears in the paragraph P–A–R–P–001. In fact, trailers are printed before the report titles if the value in CURRENT-LINE is equal to fifty-five.

Unlike the report title logic, the report trailer function is accomplished with only two COBOL instructions, because our report trailer is only one printed line preceded by a blank line. Not all report trailers are so brief. It is quite possible to have a series of report trailers whose total size exceeds that of the report titles. In that case, we would be just as concerned that the COBOL statements to produce report trailers appeared twice in our PROCEDURE DIVISION. These instructions would substantially increase the size of our COBOL program.

```
       SKIP3
       PRINT-TITLES SECTION.
       P-TI-001.
      *
      *    ****    THIS SECTION PRINTS THE TITLES THAT ARE TO
      *            APPEAR AT THE TOP OF EACH PAGE OF THE REPORT.
           DISPLAY BLANK-LINE.
           DISPLAY BLANK-LINE.
           ADD 1 TO PAGE-NUMBER.
           DISPLAY TITLE-1.
           DISPLAY TITLE-2.
           DISPLAY TITLE-3.
           DISPLAY BLANK-LINE.
           DISPLAY TITLE-4.
           DISPLAY BLANK-LINE.
           DISPLAY BLANK-LINE.
           DISPLAY TITLE-5.
           DISPLAY TITLE-6.
           DISPLAY TITLE-7.
           DISPLAY BLANK-LINE.
           MOVE 11 TO CURRENT-LINE.
```

Figure 8.3
Subroutine to print report titles

Creating Subroutines

In many programming languages, including COBOL, common functions can be grouped at just one point in the program. Every time that function is *invoked,* control of the program is transferred to that function, the function performs its task, and control then returns to the *point of invocation.*

Invoke
To cause a subroutine to come into action and to carry out its assigned functions.

Point of Invocation
The location in the program where a subroutine is invoked. Usually, when the subroutine completes execution, control goes to the statement after the invocation.

One benefit of the subroutine concept is that the actual programming statements for the function appear in the program only once. If the statements used to produce our report titles were grouped in a subroutine, then our COBOL program would be almost fifteen statements shorter.

Shorter COBOL programs are desirable. Shorter programs are easier to read and comprehend than are longer programs designed to produce the same reports. Programs that are easier to comprehend are developed more quickly, are usually more reliable, and are easier to repair and enhance.

Defining Subroutines

Subroutine definition is easily accomplished in COBOL. To define a subroutine, take the COBOL statements that form the common function and create a new section in the PROCEDURE DIVISION that identifies the function. For instance, figure 8.3 illustrates a subroutine that is used to print report titles. Because we have created a print titles subroutine, these same COBOL statements need not appear in the original two locations noted earlier. As a result, we will modify our PROCEDURE DIVISION as shown in figure 8.4. However, our newest version of the PROCEDURE DIVISION is not quite as short as was originally promised for several reasons. First, when creating new sections in the PROCEDURE DIVISION, we must add documenting comments to explain the function of these new sections. These comments require punched cards in our program deck. Second, one subroutine (PRINT-TRAILERS) does not save any keypunching whatsoever. As we mentioned earlier, however, if the trailer messages to be printed were more numerous, a savings would have been possible.

Figure 8.4
Revised PROCEDURE DIVISION using subroutines

Page 1 of 4

```
       PROCEDURE DIVISION.
           SKIP1
       MAIN-CONTROL SECTION.
       M-C-001.
      *
      *    ****    PRINT THE REPORT TITLES THAT MUST APPEAR AT THE
      *            BEGINNING OF THE REPORT.
           PERFORM PRINT-TITLES.
           SKIP2
       M-C-002.
      *
      *    ****    THIS SEGMENT OF THE PROGRAM SERVES TO CONTROL THE
      *            PRINT AND READ PUMP SECTION THAT FOLLOWS.  TO
      *            START THE PUMP ONE CARD IMAGE IS READ AND THEN
      *            PUMP IS EXECUTED.  WHEN THE BLANK CARD IS FOUND,
      *            THE PUMP TERMINATES.
           ACCEPT TIME-CARD-INPUT.
           PERFORM PRINT-AND-READ-PROCESS
               UNTIL
                   TIME-CARD-INPUT = SPACES.
           SKIP2
       M-C-003.
      *
      *    ****    NOW THAT THE PRINTING/READING PROCESS IS OVER,
```

Page 2 of 4

```
      *            COMPLETE THE LAST PAGE OF THE REPORT BY PRINTING
      *            THE FINAL SET OF TRAILERS.
           PERFORM PRINT-TRAILERS.
           STOP RUN.
           SKIP3
       PRINT-AND-READ-PROCESS SECTION.
       P-A-R-P-001.
      *
      *    ****    THIS SEGMENT CHECKS TO SEE IF PAGE OVERFLOW
      *            WILL OCCUR WITH THE PRINTING OF ONE ADDITIONAL
      *            LINE.  IF SO, THEN TRAILERS AND TITLES ARE
      *            PRINTED WHERE APPROPRIATE.
           IF CURRENT-LINE = 55
               THEN
                   PERFORM PRINT-TRAILERS
                   PERFORM PRINT-TITLES.
           SKIP2
       P-A-R-P-002.
      *
      *    ****    THE LAST RECORD IMAGE READ IS REFORMATTED
      *            AND PRINTED.  THE NEXT RECORD IMAGE TO BE
      *            PROCESSED IS READ.
           ADD 1 TO CURRENT-LINE.
           MOVE CORRESPONDING TIME-CARD-INPUT TO
```

Invoking COBOL Subroutines

The bold-faced statements in the revised PROCEDURE DIVISION are similar to the iterative statement that controls the print-and-read process. The only difference is the lack of an iteration specification. The following COBOL statements are equivalent statements and would produce identical outcomes.

```
PERFORM PRINT-TITLES.
```

and

```
PERFORM PRINT-TITLES
    1 TIMES.
```

```
                        TIME-CARD-OUTPUT.
           DISPLAY TIME-CARD-OUTPUT.
           ACCEPT  TIME-CARD-INPUT.
           SKIP3
       PRINT-TITLES SECTION.
       P-TI-001.
      *
      *    ****    THIS SECTION PRINTS THE TITLES THAT ARE TO
      *            APPEAR AT THE TOP OF EACH PAGE OF THE REPORT.
           DISPLAY BLANK-LINE.
           DISPLAY BLANK-LINE.
           ADD 1 TO PAGE-NUMBER.
           DISPLAY TITLE-1.
           DISPLAY TITLE-2.
           DISPLAY TITLE-3.
           DISPLAY BLANK-LINE.
           DISPLAY TITLE-4.
           DISPLAY BLANK-LINE.
           DISPLAY BLANK-LINE.
           DISPLAY TITLE-5.
           DISPLAY TITLE-6.
           DISPLAY TITLE-7.
           DISPLAY BLANK-LINE.
           MOVE 11 TO CURRENT-LINE.
```



```
           SKIP3
       PRINT-TRAILERS SECTION.
       P-TR-001.
      *
      *    ****    THIS SECTION PRINTS THE TRAILERS WHICH APPEAR
      *            AT THE BOTTOM OF EACH PAGE OF THE PRINTED
      *            REPORT.
           DISPLAY BLANK-LINE.
           DISPLAY TRAILER-1.
```

Both versions of the PERFORM statement are new to you. However, the second version is very similar to the iterative statements used since the beginning of the text. Here it is assumed that the programmer knows exactly how many times the section entitled PRINT-TITLES is to be invoked: once. Consequently, when the computer encounters the PERFORM statement, the following series of actions takes place:

1. The computer notes the location of the COBOL instruction that immediately follows the PERFORM statement.
2. Control of the computer proceeds from the PERFORM statement directly to the section whose name appears as part of the PERFORM.
3. The section named is executed instruction by instruction from beginning to end.
4. When the end of the section has been reached, the computer returns to the instruction immediately following the PERFORM statement that directed control to the section.

Statement Format

```
B
↓
PERFORM sectionname [integer TIMES].
```

The Benefits of Subroutines

The only advantage associated with subroutines shown thus far is to make our lengthy COBOL programs shorter. A related benefit also merits discussion, however.

Program maintenance is a thankless task. Typically, the individual assigned to maintain a computer program is not the person who created the program. Reading and comprehending someone else's computer programs often represents a less than ideal set of circumstances. Although report programs can all be structured in the same basic way, they usually are not. Programmers prefer to experiment and be creative when structuring their programs. Even uniformly structured programs can be very complex and difficult to comprehend.

By creating subroutines, we can remove many COBOL statements from the main flow of program control. The statement

```
PERFORM PRINT-TITLES.
```

tells the reader exactly what is to take place next—report titles are to be printed. To find out exactly how those titles are to be printed, the section entitled PRINT-TITLES can be located to examine its contents. On the other hand, to verify that the main control portion of the program's PROCEDURE DIVISION will print titles, only the appropriate PERFORM statement must be located.

Without these subroutines, the coding to print titles appears in the main control portion of the program. This makes this portion of the program much longer and, correspondingly, more difficult to read. It is likely to be more difficult to verify when and how titles are to be printed.

Compare our most recent PROCEDURE DIVISION in figure 8.5, with the one shown at the end of chapter 7. Which PROCEDURE DIVISION is easier to understand?

Figure 8.5
Final version of COBOL report program using subroutines

Page 1 of 7

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  RPT0801.
     SKIP2
 ENVIRONMENT DIVISION.
     SKIP2
 DATA DIVISION.
 WORKING-STORAGE SECTION.
*
*    ****  ALL PROGRAM COUNTERS
*
 01  CURRENT-LINE                PICTURE IS 9(2) VALUE ZERO.
     SKIP2
*
*    ****  ALL REPORT TITLES AND TRAILERS
*
 01  TITLE-1.
     02 FILLER                   PICTURE IS X(50) VALUE SPACES.
     02 FILLER                   PICTURE IS X(62) VALUE
     "ACME MACHINE COMPANY".
     02 FILLER                   PICTURE IS X(05) VALUE
     "PAGE-".
     02 PAGE-NUMBER              PICTURE IS 9(03) VALUE 1.
     SKIP1
 01  TITLE-2                     PICTURE IS X(120) VALUE
```

Figure 8.5
(continued)

Page 2 of 7

```
           "                                            LEISURE AND SPORTS
      -    " PRODUCTS DIVISION".
           SKIP1
       01  TITLE-3                 PICTURE IS X(120) VALUE
           "                                                  MINNEAPOLIS
      -    " SALES OFFICE".
           SKIP1
       01  TITLE-4                 PICTURE IS X(120) VALUE
           "                                                   WEEKLY PAY
      -    "ROLL REPORT".
           SKIP1
       01  TITLE-5                 PICTURE IS X(120) VALUE
           "         SOCIAL          EMPLOYEES                  EMPLOYEES
      -    "                   EMPLOYEES    EMP     EMP     HRS    HRLY".
           SKIP1
       01  TITLE-6                 PICTURE IS X(120) VALUE
           "         SECURITY          NAME                        STREET
      -    "                   CITY         STE     ZPCDE   WKD    WAGE".
           SKIP1
       01  TITLE-7                 PICTURE IS X(120) VALUE
           "         NUMBER                                       ADDRESS
      -    "                   ADDRESS      CDE".
           SKIP1
       01  TRAILER-1               PICTURE IS X(120) VALUE
```

Page 3 of 7

```
           "                                                   COMPANY CO
      -    "NFIDENTIAL".
           SKIP1
       01  BLANK-LINE              PICTURE IS X(120) VALUE SPACES.
           SKIP2
      *    **** ALL OTHER VARIABLES
      *
       01  TIME-CARD-INPUT.
           02 SOCIAL-SECURITY      PICTURE IS X(11).
           02 EMPLOYEE-NAME        PICTURE IS X(25).
           02 EMPLOYEE-STREET      PICTURE IS X(20).
           02 EMPLOYEE-CITY        PICTURE IS X(10).
           02 EMPLOYEE-STATE       PICTURE IS X(02).
           02 EMPLOYEE-ZIPCODE     PICTURE IS 9(05).
           02 EMPLOYEE-HOURS       PICTURE IS 9(02).
           02 EMPLOYEE-WAGE        PICTURE IS 9(04).
           02 FILLER               PICTURE IS X(01).
           SKIP1
       01  TIME-CARD-OUTPUT.
           02 FILLER               PICTURE IS X(07) VALUE SPACES.
           02 SOCIAL-SECURITY      PICTURE IS X(11).
           02 FILLER               PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-NAME        PICTURE IS X(25).
           02 FILLER               PICTURE IS X(04) VALUE SPACES.
```

Figure 8.5
(continued)


```
           02 EMPLOYEE-STREET          PICTURE IS X(20).
           02 FILLER                   PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-CITY            PICTURE IS X(10).
           02 FILLER                   PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-STATE           PICTURE IS X(02).
           02 FILLER                   PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-ZIPCODE         PICTURE IS 9(05).
           02 FILLER                   PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-HOURS           PICTURE IS 9(02).
           02 FILLER                   PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-WAGE            PICTURE IS 9(04).
           02 FILLER                   PICTURE IS X(06) VALUE SPACES.
           SKIP3
       PROCEDURE DIVISION.
           SKIP1
       MAIN-CONTROL SECTION.
       M-C-001.
      *
      *    ****     PRINT THE REPORT TITLES THAT MUST APPEAR AT THE
      *             BEGINNING OF THE REPORT.
           PERFORM PRINT-TITLES.
           SKIP2
       M-C-002.
      *
```



```
      *    ****   THIS SEGMENT OF THE PROGRAM SERVES TO CONTROL THE
      *           PRINT AND READ PUMP SECTION THAT FOLLOWS.  TO
      *           START THE PUMP ONE CARD IMAGE IS READ AND THEN
      *           PUMP IS EXECUTED.  WHEN THE BLANK CARD IS FOUND,
      *           THE PUMP TERMINATES.
           ACCEPT TIME-CARD-INPUT.
           PERFORM PRINT-AND-READ-PROCESS
                  UNTIL
                       TIME-CARD-INPUT = SPACES.
           SKIP2
       M-C-003.
      *
      *    ****   NOW THAT THE PRINTING/READING PROCESS IS OVER,
      *           COMPLETE THE LAST PAGE OF THE REPORT BY PRINTING
      *           THE FINAL SET OF TRAILERS.
           PERFORM PRINT-TRAILERS.
           STOP RUN.
           SKIP3
       PRINT-AND-READ-PROCESS SECTION.
       P-A-R-P-001.
      *
      *    ****   THIS SEGMENT CHECKS TO SEE IF PAGE OVERFLOW
      *           WILL OCCUR WITH THE PRINTING OF ONE ADDITIONAL
      *           LINE.  IF SO, THEN TRAILERS AND TITLES ARE
```

Summary

COBOL programs can be quite long and difficult to read and comprehend. Comprehension is not only important in the initial design and construction of COBOL programs but in maintenance considerations as well. Hopefully, standard program designs in our report programs allows greater insight into how those programs carry out their tasks. Further, the use of subroutines is an additional aid.

Program subroutines allow us to relocate sections of COBOL statements, taking them out of the important control processes of our programs where their inclusion only serves to clutter the program's appearance. By invoking these subroutines using the COBOL PERFORM statement, the reader can determine the function that is to be performed without losing track of the overall program task.

Page 6 of 7

```
      *          PRINTED WHERE APPROPRIATE.
           IF CURRENT-LINE = 55
               THEN
                   PERFORM PRINT-TRAILERS
                   PERFORM PRINT-TITLES.
           SKIP2
       P-A-R-P-002.
      *
      *    ****  THE LAST RECORD IMAGE READ IS REFORMATTED
      *          AND PRINTED.  THE NEXT RECORD IMAGE TO BE
      *          PROCESSED IS READ.
           ADD 1 TO CURRENT-LINE.
           MOVE CORRESPONDING TIME-CARD-INPUT TO
                              TIME-CARD-OUTPUT.
           DISPLAY TIME-CARD-OUTPUT.
           ACCEPT TIME-CARD-INPUT.
           SKIP3
       PRINT-TITLES SECTION.
       P-TI-001.
      *
      *    ****  THIS SECTION PRINTS THE TITLES THAT ARE TO
      *          APPEAR AT THE TOP OF EACH PAGE OF THE REPORT.
           DISPLAY BLANK-LINE.
           DISPLAY BLANK-LINE.
```

Page 7 of 7

```
           ADD 1 TO PAGE-NUMBER.
           DISPLAY TITLE-1.
           DISPLAY TITLE-2.
           DISPLAY TITLE-3.
           DISPLAY BLANK-LINE.
           DISPLAY TITLE-4.
           DISPLAY BLANK-LINE.
           DISPLAY BLANK-LINE.
           DISPLAY TITLE-5.
           DISPLAY TITLE-6.
           DISPLAY TITLE-7.
           DISPLAY BLANK-LINE.
           MOVE 11 TO CURRENT-LINE.
           SKIP3
       PRINT-TRAILERS SECTION.
       P-TR-001.
      *
      *    ****  THIS SECTION PRINTS THE TRAILERS WHICH APPEAR
      *          AT THE BOTTOM OF EACH PAGE OF THE PRINTED
      *          REPORT.
           DISPLAY BLANK-LINE.
           DISPLAY TRAILER-1.
```

Another benefit of program subroutines is that they may reduce overall program size. Program functions to be executed more than once in the program can now be accessed from those points using the PERFORM statement. In each instance, control passes to the desired function, the function is executed, and control returns to the point of invocation.

At this point, our COBOL report program appears as shown in figure 8.5.

Questions for Review

1. As a future maintenance programmer, what do you find disquieting about the prospect of reading a program written without an accepted structuring of its logical functions? (Hint: The report program is designed differently from those described thus far.)
2. Why are smaller programs easier to read than those which may span many, many pages? In what manner does the subroutine assist in making programs smaller?
3. Why didn't the report program described in this chapter markedly shrink in size when the report title statements were regrouped into one subroutine invoked by a PERFORM statement? Under what circumstances would the savings in program size have been greater?
4. Of what importance is the choice of section or subroutine name in the design of a COBOL program?

Problems

1. Pretend that the report program shown in figure 8.5 could contain only one DISPLAY statement in the entire PROCEDURE DIVISION. Construct such a PROCEDURE DIVISION—one that uses only a single DISPLAY. (Hint: Create a subroutine named PRINT-LINE which is performed every time a line of printed output is desired.) Would this alteration make the report program shorter or easier to comprehend?

Programming Assignments

1. Revise the program on the Quatraplex Systems assignment from chapter 7 to use a separate subroutine for each of the following program functions:
   a. acquiring and reformatting both the report date and time,
   b. printing report titles,
   c. printing report trailers, and
   d. computing both the elapsed days and days late data values.
2. Revise the program on the Columbia Concrete Products assignment from chapter 7 to use a separate subroutine for each of the following program functions:
   a. acquiring and reformatting both the report date and time,
   b. selecting only records that apply to the U.S. Army Corps of Engineers,
   c. printing report titles, and
   d. printing report trailers.

Sequential File Processing
Control Breaks and Subtotals

9

Chapter Outline

Report programs frequently list data applying to different departments within an organization in the same report. In most circumstances, the data for the first department will be printed in its entirety, followed by all of the data for the second department, and so on. In such cases, department totals are required in addition to overall corporate totals. This chapter illustrates the alterations to the report program logic required to support organizational hierarchies. A payroll report program application illustrates this point.

Organizations as Hierarchies

One of the largest expenses in most organizations is the cost of employing people. Thus a payroll report, while listing the payroll for the entire company, is likely to be organized by division. The first segment of the report will concern itself with the payroll costs of the first division in the organizational structure, the second segment with the second division, and so on.

Typically, many organizations assign each employee a payroll identification number in addition to the employee's Social Security number. These payroll numbers are designed to identify the division within the company to which a particular person is assigned. Company-specified payroll numbers fulfill a need that Social Security numbers cannot. Social Security numbers are allocated by the federal government as they are requested. They cannot identify in which division of an organization an employee belongs. (Because the word DIVISION is reserved in COBOL and may not be used as a variable name, the word BUREAU will be used in this text to denote a division of a corporation.) Assigned payroll identification numbers, however, can do just that.

Assume that a hypothetical organization has assigned eleven digit payroll numbers to each of its employees. Each payroll number is made up of three parts. The first part denotes the bureau in which the employee works. The second denotes the employee's department or activity within the bureau. Finally, the third portion uniquely identifies the particular employee within the bureau and department. For instance,

```
AA-BB-CCCCC
                        AA-bureau (01 thru 99)
                        BB-department (01 thru 99)
                        CCCCC-person (00001 thru 99999)
```

When the file of all employees is created, the employees are arranged first by bureau, second by department within each bureau, and finally by ascending sequence number within each department. Upon reading the payroll input file from beginning to end, the sequence of employees shown in figure 9.1 is likely to occur. Note how department

Figure 9.1
Organization of payroll input file

Bureau number	Department number	Employee number
01	01	00001
01	01	00002
01	01	00003
.	.	.
01	01	99999
01	02	00001
.	.	.
01	02	99999
.	.	.
.	.	.
01	99	99999
02	01	00001
.	.	.
.	.	.
.	.	.
99	99	99999

01 employees (of bureau 01) come before those of department 02. Also, note how all of the employees in bureau 01 come before those of bureau 02. If we are to produce subtotals for each department within each bureau, and for each bureau as well, then our report program logic must be modified once more. We must now check each input record read, and compare that record with the prior record to see if the department number has changed. If the department number has changed, then the department totals for the just-ended department must be produced. Then the numeric variables in which those totals were produced must be reset to zero for accumulation of the data for the new department.

A change in the bureau from one input record to the next is somewhat more complex. First, you must realize that a change in bureaus is also a change in departments. Remember, departments belong to bureaus. So, if a new bureau is encountered, then the last bureau has been completely processed. Stated somewhat differently, the last department belonging to the previous bureau has come to an end. We are now examining the first employee in the current bureau's first department. As a result, the process just described for a change in departments must occur for bureaus as well. Once the department subtotals have been printed and the respective numeric variables reset, then the same procedure must be carried out for the bureau subtotals. After all, bureaus, like departments, have subtotals.

The most complex *control break* occurs when we have exhausted the input file. At this point we have first a department break, followed by a bureau break, followed by a company break. A company break is the time to print the totals accumulated for the entire company. See the summary shown in figure 9.2.

Control Break
A change in the value of a key, or important, field from one input record to the next.

Revised Program Logic

Our overall program logic, as expressed in pseudo-coding, was last modified to number the pages produced by the report program. Figure 9.3 shows the program logic for report programs developed thus far. Our program logic must now be modified to check for control breaks every time a new record image is read from the input file. One of these alterations must occur in the section of statements identified as the "print-and-read process."

Control break	Processing task
New employee	(1) add numeric values being tallied from input record to department (sub)totals.
New department	(1) print the subtotals for the just completed department. (2) add department subtotals to the bureau subtotals (3) reset department subtotals to zero.
New bureau	(1) print the subtotals for the just completed department. (2) add department subtotals to the bureau subtotals (3) reset department subtotals to zero. (4) print subtotals for the just completed bureau. (5) add bureau subtotals to the corporate totals (6) reset bureau subtotals to zero.
End of data	(1) print the subtotals for the just completed department. (2) add department subtotals to the bureau subtotals (3) print the subtotals for the just completed bureau (4) add bureau subtotals to the corporate totals (5) print the corporate totals

Figure 9.2
Summary of control breaks and processing tasks

Figure 9.3
Report program logic without control breaks

```
Statement    Pseudo-coded statement
 number

  1              perform page titles
  2              read very first record from input file
  3              perform print and read process
  3a                 until
  3b                     record read is blank
  4              perform page trailer
  5              stop execution

  6        print and read process
  7              if page overflow
  7a                 then
  7b                     reset line counter
  7c                     perform page trailer
  7d                     perform page titles
  8              increment line counter by 1
  9              move input record to output record
 10              print reformatted record
 11              read very next record

 12          page titles
 13              print titles separated by blank lines

 14          page trailers
 15              increment page number by 1
 16              print blank line followed by trailer
```

Figure 9.4
Proposed location for control break monitoring

```
  6        print and read process
  7              if page overflow
  7a                 then
  7b                     reset line counter
  7c                     perform page trailer
  7d                     perform page titles
  8              increment line counter by 1
  9              move input record to output record
 10              print reformatted record
Check here?———————>
 11              read very next record
```

At present the first activity within this particular group of pseudo-coded statements concerns itself with page overflow. Following the page overflow logic, our program proceeds to reformat the input record last read, print that record, and then read the next record from the input file.

If we do not check for a control break before reformatting and printing the most recently read input record, then we risk printing a record in our current department-bureau group that may be of (1) a different department, or (2) a different bureau as well as a different department. In other words, it seems illogical to check for a control break in the location indicated in figure 9.4. If we check for a control break at this particular location in our report program logic, the sequence of events would be:

1. check for, and respond to, a possible page overflow condition.
2. reformat the most recently read record from the input file. Remember, this record could belong to a different department.
3. print this most recently read record, now reformatted.
4. if a control break is detected, perform the activities described earlier in this chapter.
5. read the very next record from the input file.

```
6         print and read process
7                 if page overflow
7a                    then
7b                          reset line counter
7c                          perform page trailer
7d                          perform page titles
```

Insert logic to find control break here →

```
8                 increment line counter by 1
9                 move input record to output record
10                print reformatted record
11                read very next record
```

Figure 9.5
Correct location for monitoring control breaks

```
1                 perform page titles
2                 read very first record from input file
3                 perform print and read process
3a                    until
3b                          record read is blank
```

Insert end of data program logic here →

```
4                 perform page trailer
5                 stop execution

6         print and read process
7                 if page overflow
7a                    then
7b                          reset line counter
7c                          perform page trailer
7d                          perform page titles
```

Insert logic to monitor for control breaks due to new departments and/or bureaus here →

```
8                 increment line counter by 1
9                 move input record to output record
10                print reformatted record
11                read very next record

12          page titles
13                print titles separated by blank lines

14          page trailers
15                increment page number by 1
16                print blank line followed by trailer
```

Figure 9.6
Control break at end of data

The correct location of the logic to check each input record for control breaks is shown in figure 9.5.

Unfortunately, inserting the pseudo-coded statements that monitor for control breaks at this point fails to take note of a very important fact—one control break occurs when the input file has been exhausted. The test already present in our program logic for the end of data occurs in the iterative statement outside of the "print-and-read-process." Examine the entire logical structure of our report program shown in figure 9.6. When the end of data occurs, the most complex summary tasks must take place. First, all of the subtotals for the current department must be produced. Second, the same task for bureaus must then occur. Finally, the end of report totals for the whole company must be produced.

Figure 9.7
Correct method of detecting control breaks

```
if bureau number is different
   then
        perform department totals
        perform bureau totals
if department number is different BUT
  bureau number is the same
   then
        perform department totals
add employee's numeric values to those
   being accumulated for this department
          .
          .
          .

bureau totals
   print bureau totals
   add bureau totals to
        corporate totals
   reset bureau totals to zero
        for next bureau

department totals
   print department totals
   add department totals to
        bureau totals
   reset department totals to zero
        for next department
```

Detecting Changes in Department-Bureau

Now that the proper locations for detecting control breaks have been found, the next issue to be addressed is what those pseudo-coded statements should look like. Figure 9.7 illustrates one possible configuration. The numeric values we are accumulating depend on the particular report our program logic is to produce. Remember, while all report programs can share a common logical structure, they report data unique to the application for which the program was constructed. Inventory reports, payroll reports, and so on can be similar in structure even though they report entirely different data. In the case of the inventory report, the values being accumulated for each input record may be quantity on hand, quantity ordered, and so on. These values would be subtotalled by inventory department (furniture or lamps or wall hangings) and by inventory bureau (home furnishings). Payroll reports might accumulate the hours worked by department, by bureau, and for the whole company.

Processing End of Data

When the end of the input file has been reached, the pseudo-coded logic to be executed might appear as

```
perform department totals
perform bureau totals
perform corporate totals
```

Final Report Program Logic

We have finally developed the logical structure of our report program to its fullest extent. This is the last time that you will be asked to modify the report program structure. Its final form is shown in figure 9.8.

Figure 9.8
Report program logic with control break modifications

```
Statement   Pseudo-coded statement
number

  1           perform page titles
  2           read very first record from input file
  3           perform print and read process
  3a              until
  3b                  record read is blank
  4           perform department totals
  5           perform bureau totals
  6           perform corporate totals
  7           perform page trailer
  8           stop execution

  9       print and read process
 10              if page overflow
 10a                then
 10b                    reset line counter
 10c                    perform page trailer
 10d                    perform page titles
 11              if bureau number different
 11a                then
 11b                    perform department totals
 11c                    perform bureau totals
 12              if department number different BUT
 12a                bureau number same
 12b                then
 12c                    perform department totals
 13              add employee's numeric values to those
 13a                being accumulated for this
                    department
 14              increment line counter by 1
 15              move input record to output record
 16              print reformatted record
 17              read very next record

 18       page titles
 19            print titles separated by blank lines

 20       page trailers
 21            increment page number by 1
 22            print blank line followed by trailer

 23       corporate totals
 24            print corporate totals

 25       bureau totals
 26           print bureau totals
 27           add bureau totals to
                  corporate totals
 28           reset bureau totals to zero
                  for next bureau

 29       department totals
 30            print department totals
 31            add department totals to
                    bureau totals
 32            reset department totals to zero
                    for next department
```

Implementation in COBOL

Key Field
An important field whose value, in the context of control breaks, indicates whether a new department or division has been encountered.

It should be apparent that detecting control breaks will not occur unless the computer has been instructed to examine the records being processed. After a record has been read, a report program must contain the appropriate means of examining a *key field,* such as employee number, to determine if a control break has occurred.

The principal means of detecting control breaks in COBOL is through the use of the IF statement. The IF statement was originally introduced in an earlier chapter. At that time it was described as an inquiry statement allowing the computer to look for a particular condition, or set of conditions, and act accordingly. For instance,

```
                    IF CURRENT-LINE = 55
                         THEN
start pathway 1->   PERFORM PRINT-TRAILERS
end   pathway 1->   PERFORM PRINT-TITLES.
pathway 2->ADD 1 TO CURRENT-LINE.
```

Here the computer is directed to inquire about the numeric value contained in a variable known as CURRENT-LINE. The relational operator (i.e., =) tests to see if the value in CURRENT-LINE is the same as the literal 55. Only two outcomes to this inquiry are possible. Either the value in CURRENT-LINE is 55, or it is not. In either case, the IF statement provides the appropriate pathways for processing.

The first processing pathway is shown by using indentation. This particular pathway is only activated if the value in CURRENT-LINE is equal to 55. Should this be the situation, then control proceeds through each of the printing operations indicated, to the start of the second pathway. The second processing pathway is defined as starting at the statement immediately following the end of the first pathway. You should recall that the first pathway is terminated when the first period after the IF verb is encountered. Thus the second pathway begins with the ADD verb.

Format of the Inquiry Condition

The phrase immediately following the IF verb is known as the inquiry condition. The inquiry condition contains two parts. First, the condition includes two items being compared. These items can be any combination of literals and variables. You can compare two literals, two variables, or a literal to a variable. It should be noted, however, that the two items being compared must possess compatible attributes.[1]

When variables are being compared to literals, it doesn't matter in which order they appear. As a result, the statements

```
IF CURRENT-LINE = 55
        or
IF 55 = CURRENT-LINE
```

are said to be equivalent. As a matter of style, the examples in this text always list the variable first and the literal second.

Statement Format

```
B
↓
IF literal operator variablename
   THEN
      statement(s).
```

Statement Format

```
B
↓
IF variablename operator literal
    THEN
        statement(s).
```

The second part of the inquiry condition is the relational operator.

Relational Operators

The only relational operator used thus far in these discussions has been the equal (=) operator. Many others are also possible. For example, another operator is the unequal (NOT =). These two operators produce opposite results in terms of which processing pathway is chosen by the computer as an outcome of the comparison operation.

Other operators include greater than (>) , less than (<), not greater than (NOT >), and not less than (NOT <). The first two operators are fairly straightforward while the last two may cause some confusion. To avoid this confusion, remember that really only three basic conditions are possible—greater than, equal, and less than. Not greater than means one of the two remaining operators: equal or less than. Not less than means one of the two remaining operators: equal or greater than.

Complex Inquiry Conditions

Often a simple inquiry condition is not sufficient. Instances in which a program is looking for two or more conditions to occur require a more complex version of the IF statement. COBOL provides two additional operators: AND and OR. The rules for their use are easy to comprehend. Examine the complex inquiry condition shown in figure 9.9. In this figure, statements one through the last are to be executed only if BOTH A is equal to B AND C is equal to D. The only way the computer can access the indented statements is if both conditions are true. If one or both of the conditions is false, then the whole inquiry is false and the indented statements cannot be executed.

Statement Format

```
B
↓           {AND}
IF condition {OR } condition ...
    THEN
        statement(s).
```

```
IF A = B
    AND
  C = D
      THEN
          statement one
          statement two
               .
               .
          last statement.
```

Figure 9.9
Complex inquiry condition

Figure 9.10
Application of complex inquiry condition in control breaks

```
INPUT record images

    10-654-28486    .....    (first record read)
    11-654-23786    .....    (second record read)

(1) if bureau number is different
       then
           perform department totals
           perform bureau totals
(2) if department number is different BUT
       bureau number is the same
        then
           perform department totals
```

An additional example illustrates this point. Our program logic for control breaks contains two inquiry statements. They are shown in figure 9.10. (Also, examine the following input records.) Recall that the first portion of these employee numbers represents the bureau to which the employee is a member. The second portion represents the employee's department within the bureau. The last portion is the actual number of the employee within the department. Because the bureau numbers are different, our program logic must realize that departments have changed as well, even though the actual department numbers may be the same. Two different departments can have the same numbers only if they belonged to different bureaus.

Because the bureaus have changed, and the departments as well, our report program must first produce the department totals for department number 654 within bureau 10. Second, the totals for bureau 10 must be produced. When control reaches the second inquiry statement, the logic that produces department totals will not be activated. This is because the department numbers are NOT different. They are the same.

The COBOL equivalent of this pseudo-coded logic is shown in figure 9.11. As you saw in figure 9.10, the division numbers of the two card images are indeed different. The first COBOL IF statement will be evaluated as being true. Consequently, the indented PERFORM statements will be executed producing the desired totals for department and bureau. The second COBOL IF statement, however, will not be evaluated as true. The reason is that the department numbers of both cards are the same. Yet the inquiry condition stipulates that the indented statement can only be executed if they are different. Since one of the two conditions is false, the entire inquiry is false. The AND operator requires that both conditions be true if the indented statements are to be executed.

What if the two card images being processed contained different bureau and department numbers? For example,

```
10-387-56249....    (first card read)
11-554-77771....    (second card read)
```

Because the bureau numbers are different, both department and bureau totals must be produced. The first COBOL IF statement guarantees these actions. What about the second COBOL IF statement?

The first of the two conditions in this second IF statement is true. The department numbers are different. What is important to remember, though, is that the bureau numbers are different, too. Further, both department and bureau totals have already been printed. To avoid having an unnecessary set of department totals printed, a qualifier condition is appended to the second IF condition. This qualifier requires that the

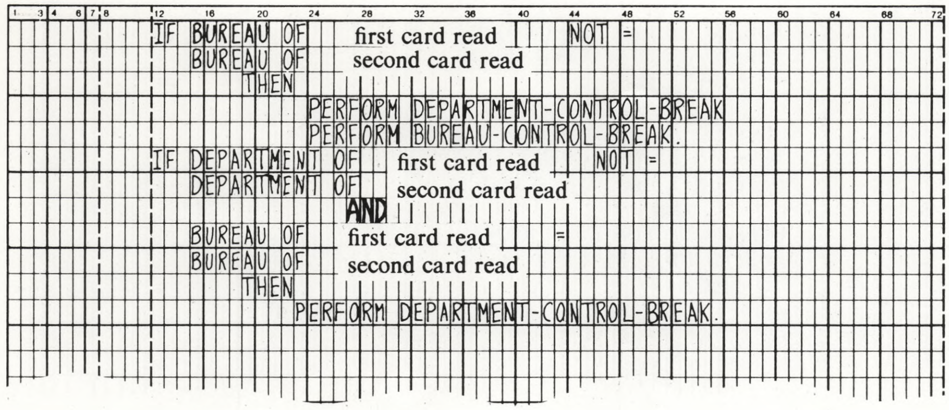


Figure 9.11
COBOL equivalent of pseduo-code complex inquiry

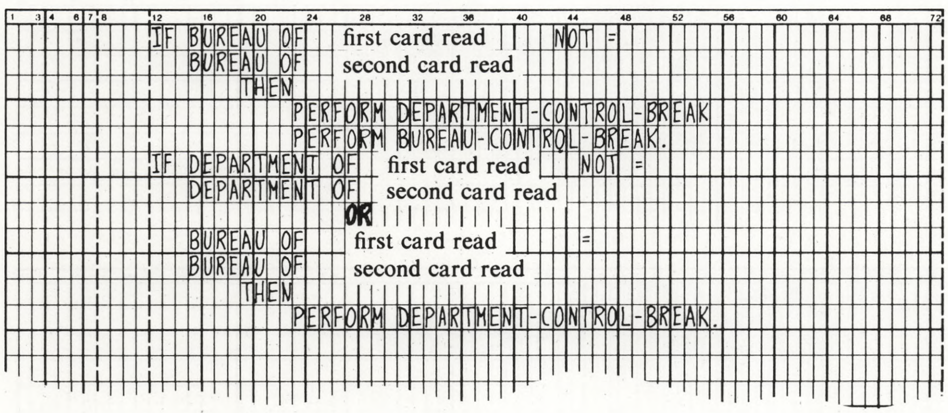


Figure 9.12
Use of OR rather than AND

bureau numbers be the same for the department totals to be printed. In this last example, the department numbers differ and the bureau numbers differ as well. Consequently, the first portion of the condition is true, but the second portion is false. The AND operator requires that both portions be true if the indented statement is to be executed. Since one of these is false, the indented statement will not be activated. Hence, a second set of department totals will not be printed.

The OR operator, unlike the AND, requires that only one of the two conditions specified be true if the indented statements are to be executed. To see the dangers that might be involved if OR is used rather than AND in the just described situation, examine the last example using OR as shown in figure 9.12. The bureau numbers and the department numbers for each record were different. Thus the first IF statement was true and the department and bureau totals were produced. The second IF statement, which was false when using the AND, is now true, and a second set of department totals is produced. Remember, the OR requires that only one of the parts of the inquiry condition be true for the entire condition to be evaluated as true and the indented statement executed.

An inquiry condition can contain multiple ANDs. However, all of the conditions must be true if the entire inquiry is to be evaluated as true and the indented statements activated. In other words, if one of those individual conditions is false, then the entire inquiry is evaluated as false and the indented statements are not activated.

An inquiry condition may also contain multiple ORs as well. Here it is required that only one part of the total conditional statement be true and the whole inquiry condition will be evaluated as true and the indented statements will be executed by the computer.

Inquiry statements can contain both ANDs and ORs. This, however, constitutes an advanced usage of the IF statement and will be discussed in a future chapter.

Figure 9.13
Monitoring control breaks in COBOL

```
       IF BUREAU OF CONTROL-INFORMATION NOT =
          BUREAU OF TIME-CARD-INPUT
             THEN
                PERFORM DEPARTMENT-CONTROL-BREAK
                PERFORM BUREAU-CONTROL-BREAK.
       IF DEPARTMENT OF CONTROL-INFORMATION NOT =
          DEPARTMENT OF TIME-CARD-INPUT
             AND
          BUREAU OF CONTROL-INFORMATION =
          BUREAU OF TIME-CARD-INPUT
             THEN
                PERFORM DEPARTMENT-CONTROL-BREAK.
```

Figure 9.14
Alternative in saving control information

```
       MOVE CORRESPONDING TIME-CARD-INPUT TO
                          TIME-CARD-OUTPUT.
       DISPLAY TIME-CARD-OUTPUT.
       MOVE CORRESPONDING TIME-CARD-INPUT TO
                          CONTROL-INFORMATION.
       ACCEPT TIME-CARD-INPUT.
```

Figure 9.15
Saving information needed for control breaks

```
       01  CONTROL-INFORMATION.
           02 BUREAU          PICTURE IS X(02).
           02 DEPARTMENT      PICTURE IS X(03).
```

A Final Note on Control Breaks

If our report program is to check for control breaks, it must have some method to compare the input record currently being processed with its predecessor. This can be accomplished in one of two ways. First the COBOL programmer defines a new record structure in the COBOL program to save the control information from the last record read. The record description, shown in figure 9.13, would appear in the WORKING-STORAGE SECTION. If this option is chosen, then the COBOL version of our pseudo-coded control break logic would be like that shown in figure 9.14. This requires that prior to reading a new time card for processing, the programmer must move the control information from the current time card into the CONTROL-INFORMATION record description. Then the new time card could be read as shown in figure 9.15. However, an additional record description such as CONTROL-INFORMATION is unnecessary. This leads to the second option.

Careful inspection of our program logic shows that the card currently being processed always resides in TIME-CARD-INPUT, while the preceding card image can be found in TIME-CARD-OUTPUT. As a result, we can avoid an additional definition like CONTROL-INFORMATION and rewrite our control break statements to appear as in figure 9.16. Whichever method is used, the programmer must solve the problem that is known as the first record read phenomenon. That is, when the report program reads its first time card record, with what will it compare it to see if a control break has occurred? Keep in mind that at this point TIME-CARD-OUTPUT doesn't contain anything because we are processing the very first record.

To solve this problem the programmer must assign a special initial value to two variables within TIME-CARD-OUTPUT. The definition of this record is reproduced in figure 9.17. The variables that must be given initial values are BUREAU and DEPARTMENT. What should those values be?

```
           IF BUREAU OF TIME-CARD-OUTPUT NOT =
               BUREAU OF TIME-CARD-INPUT
                   THEN
                       PERFORM DEPARTMENT-CONTROL-BREAK
                       PERFORM BUREAU-CONTROL-BREAK.
           IF DEPARTMENT OF TIME-CARD-OUTPUT NOT =
               DEPARTMENT OF TIME-CARD-INPUT
                   AND
               BUREAU OF TIME-CARD-OUTPUT =
               BUREAU OF TIME-CARD-INPUT
                   THEN
                       PERFORM DEPARTMENT-CONTROL-BREAK.
```

Figure 9.16
Alternative in saving control information

```
        01  TIME-CARD-OUTPUT.
            02 FILLER                 PICTURE IS X(06) VALUE SPACES.
            02 PAYROLL-NUMBER.
------>        03 BUREAU              PICTURE IS X(02).
               03 FILLER              PICTURE IS X(01) VALUE "-".
------>        03 DEPARTMENT          PICTURE IS X(03).
               03 FILLER              PICTURE IS X(01) VALUE "-".
               03 EMPLOYEE-ID         PICTURE IS X(05).
            02 FILLER                 PICTURE IS X(04) VALUE SPACES.
            02 EMPLOYEE-NAME          PICTURE IS X(25).
            02 FILLER                 PICTURE IS X(04) VALUE SPACES.
            02 EMPLOYEE-STREET        PICTURE IS X(20).
            02 FILLER                 PICTURE IS X(04) VALUE SPACES.
            02 EMPLOYEE-CITY          PICTURE IS X(10).
            02 FILLER                 PICTURE IS X(04) VALUE SPACES.
            02 EMPLOYEE-STATE         PICTURE IS X(02).
            02 FILLER                 PICTURE IS X(04) VALUE SPACES.
            02 EMPLOYEE-ZIPCODE       PICTURE IS 9(05).
            02 FILLER                 PICTURE IS X(04) VALUE SPACES.
            02 EMPLOYEE-HOURS         PICTURE IS 9(02).
            02 FILLER                 PICTURE IS X(04) VALUE SPACES.
            02 EMPLOYEE-WAGE          PICTURE IS 9(04).
            02 FILLER                 PICTURE IS X(06) VALUE SPACES.
```

Figure 9.17
TIME-CARD-OUTPUT with special values for control

A typical response to this last question is usually the values 01 and 001, respectively. Unfortunately, this particular choice of values is based on the assumption that the very first time card must contain a 01 for its bureau and an 001 for its department. This may not always be the case. If they ever abolished bureau 01 or department 001, the program would have to be altered. Another alternative avoids such data dependencies. What if these two variables were assigned values greater than any department or bureau could ever be assigned? Note the control break statements in figure 9.18. Compare these COBOL statements with the last version. The difference lies in the ordering of the variables in the inquiry condition and the use of a different relational operator.

In this most recent version, a control break will occur only if the incoming control values are greater than those found in the last card image. If BUREAU and DEPARTMENT in TIME-CARD-OUTPUT are assigned the values of 99 and 999, respectively, the first time card read is likely to be less than, or not greater than, these initial values. Hence no control break will occur. Therefore, the first card phenomenon is (almost) solved.

What if the first time card read does contain the bureau number 99? A safeguard might be to assign BUREAU and DEPARTMENT in TIME-CARD-OUTPUT the initial value of HIGH-VALUES, as shown in figure 9.19. HIGH-VALUES is greater than the largest numeric values that could possibly occur in a time card. We have finally found the protection desired.[2]

Figure 9.18
Alternative in monitoring control breaks

```
       IF BUREAU OF TIME-CARD-INPUT >
          BUREAU OF TIME-CARD-OUTPUT
             THEN
                PERFORM DEPARTMENT-CONTROL-BREAK
                PERFORM BUREAU-CONTROL-BREAK.
       IF DEPARTMENT OF TIME-CARD-INPUT >
          DEPARTMENT OF TIME-CARD-OUTPUT
                AND
          BUREAU OF TIME-CARD-INPUT =
          BUREAU OF TIME-CARD-OUTPUT
             THEN
                PERFORM DEPARTMENT-CONTROL-BREAK.
```

Figure 9.19
HIGH-VALUES to avoid initial control break

```
        01  TIME-CARD-OUTPUT.
            02  FILLER                 PICTURE IS X(06) VALUE SPACES.
            02  PAYROLL-NUMBER.
------>         03  BUREAU             PICTURE IS X(02) VALUE HIGH-VALUES.
                03  FILLER             PICTURE IS X(01) VALUE "-".
------>         03  DEPARTMENT         PICTURE IS X(03) VALUE HIGH-VALUES.
                03  FILLER             PICTURE IS X(01) VALUE "-".
                03  EMPLOYEE-ID        PICTURE IS X(05).
            02  FILLER                 PICTURE IS X(04) VALUE SPACES.
            02  EMPLOYEE-NAME          PICTURE IS X(25).
            02  FILLER                 PICTURE IS X(04) VALUE SPACES.
            02  EMPLOYEE-STREET        PICTURE IS X(20).
            02  FILLER                 PICTURE IS X(04) VALUE SPACES.
            02  EMPLOYEE-CITY          PICTURE IS X(10).
            02  FILLER                 PICTURE IS X(04) VALUE SPACES.
            02  EMPLOYEE-STATE         PICTURE IS X(02).
            02  FILLER                 PICTURE IS X(04) VALUE SPACES.
            02  EMPLOYEE-ZIPCODE       PICTURE IS 9(05).
            02  FILLER                 PICTURE IS X(04) VALUE SPACES.
            02  EMPLOYEE-HOURS         PICTURE IS 9(02).
            02  FILLER                 PICTURE IS X(04) VALUE SPACES.
            02  EMPLOYEE-WAGE          PICTURE IS 9(04).
            02  FILLER                 PICTURE IS X(06) VALUE SPACES.
```

Summary

Report programs often must produce subtotals as well as totals. These intermediate results are typically printed when an input record being processed is from a different unit or sub-unit within the organization. When this change occurs, we print the totals accumulated for the previous unit or sub-unit and start accumulating these same totals for the new unit or sub-unit. These changes are called control breaks.

Organizations may be composed of bureaus. Bureaus may consist of departments, departments of teams, etc. The list may go on to describe an elaborately structured organization for which subtotals are desired for each level. Control breaks that occur at the lowest level are the easiest to accommodate. Only the totals for that level need to be printed. The problems arise when control breaks occur higher in the organization structure. A change in bureaus requires that first department totals must be produced, then bureau totals must be printed.

To accommodate this kind of processing function, our report program must be modified. First, the record description that is to hold the reformatted version of the input record, for purposes of printing, must be assigned initial values for those variables that hold control information. Further, our control monitoring statements must be constructed to detect a change in these control values.

Notes

1. Not all attributes must agree in all instances. You are encouraged to consult the computer manufacturer's reference manuals to determine which combinations are permitted as well as prohibited.
2. HIGH-VALUES are actually hexidecimal FFs, which, in the EBCDIC collating sequence, are higher than the number 9 in decimal which is F9 in hexidecimal.

Questions for Review

1. Why must the input records be sorted in ascending sequence by payroll number if the control break logic in a report program is to function properly?
2. Why is it that when checking for a change in departments from one record to the next, you must also see if the bureaus have changed? What might happen if the bureau check were omitted from the inquiry statement that also checked to see if a change in department had occurred?
3. Why must the program check to see if a bureau change occurred before checking to see if a department change occurred? Can you formulate a general rule that describes in what order control breaks should be checked?
4. Examine the following partial time card images. As they are arranged, could a report program produce department, bureau, and corporate totals? Why not? What needs to be done if our report program logic is to function properly?

```
05-01-00004.....
01-05-00001.....
01-05-00002.....
02-02-00012.....
05-01-00003.....
```

Problems

1. Figure 9.18 shows the testing for control breaks in our payroll program, starting at the top of the organization's structure (i.e., bureau) and proceeding to the lowest level in the organization (i.e., department). Revise the COBOL statements to start the testing at the department level.
2. What, if anything, is wrong with the following control break logic?

```
IF DEPARTMENT OF TIME-CARD-INPUT NOT =
   DEPARTMENT OF TIME-CARD-OUTPUT
    THEN
        PERFORM DEPARTMENT-CONTROL-BREAK.
IF BUREAU OF TIME-CARD-INPUT NOT =
   BUREAU OF TIME-CARD-OUTPUT
    THEN
        PERFORM BUREAU-CONTROL-BREAK.
```

Programming Assignments

1. The HI-FI Hut, an east coast stereo equipment chain, has historically evaluated the performance of each of its units by comparing gross profits only. After all, this single indication of performance was most readily available from the chain's older batch accounting system. Recently, however, the firm's data processing facility installed on-line, interactive systems so that each sale is recorded as it is made. This allows for better inventory maintenance and record keeping. For the first time, HI-FI Hut's administrative offices have access to individual sales invoices. The new on-line, interactive system now makes it possible to evaluate each store's performance by department (i.e., receivers, speakers, etc.) as well as by total sales volume. The report program that will produce this new report will process an input file where each eighty-column record will appear as follows:

Record Position	*Data Type*	*Variable*
01–06	N	Invoice number
07–09	N	Store number
10–11	N	Department number
12–17	AN	Stock code
18–24	N	Price (dollars and cents)
25–80	–	Unused

   Each record above represents an item sold in a particular HI-FI Hut store. The records are ordered by store number, within store by department, and within department by invoice. The report program output should appear like that shown in figure 9.20.

2. The American University conducts registration for its fall, spring, and summer semesters three times each year. During each registration period students complete a brief course selection form for each course to which they desire to be admitted. Each form appears as follows:

Record Position	*Data Type*	*Variable*
01–09	N	Student's Social Security number
10–11	N	Department number of course chosen
12–14	N	Course number
15–16	N	Section of course chosen
17–80	–	Unused

   A file of these registration records is arranged by department, within department by course/section, and within course/section by student. A report program that produces the class rosters, class, department, college, and university totals is illustrated in figure 9.21. To compute college subtotals you need to know the following:

Department Number(s)	*College*
01	Honors Program
02–49	College of Arts & Sciences
50–70	College of Public & International Affairs
71–75	College of Business Administration
76–99	Miscellaneous

Figure 9.20
HI-FI Hut sales report

PROGRAMMER ____________

PROBLEM ____________

DATE ____________

REPORT LAYOUT FORM

PAGE ____ OF ____

```
                              HI-FI HUT, INC
                      NATIONAL SALES REPORTING SYSTEM
                          SALES ANALYSIS REPORT

     STORE       INVOICE       DEPARTMENT      STOCK
     NUMBER      NUMBER          NUMBER        CODE          PRICE

      341        100023            01          A432C1        14999
      341        100023            01          A432C2        24999
      341        100025            01          A331B3        84599
  ** DEPARTMENT 01 TOTALS **                                124597

      341        100023            02          B33412         9999
      341        100025            02          B32220        12099
      341        100028            02          B12001         7999
  ** DEPARTMENT 02 TOTALS **                                 30097
  ** STORE 341 TOTALS **                                    154694

      342        000081            01          A432C1        14999

  ** STORE 342 TOTALS **                                    134897

  ** CORPORATE SALES TOTALS **                            24310899

                              PAGE-XXX
```

Figure 9.21
The American University enrollment report

PROGRAMMER ______
PROBLEM ______
DATE ______

REPORT LAYOUT FORM

PAGE ____ OF ____

```
                                       THE AMERICAN UNIVERSITY                              ROSTER 1022
                                          CLASS ROSTER FOR                                  PAGE 1
                                             40.280-01

                SOC SEC NBR       STUDENT'S NAME                        COMMENTS
             01 215-50-3145       ALTER, AQ
             02 197-42-1508       BASS, SD
             25 434-11-2433       ZOE, AB
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                                       THE AMERICAN UNIVERSITY                              ROSTER 1023
                                          CLASS ROSTER FOR                                  PAGE 1
                                             40.280-02

                SOC SEC NBR       STUDENT'S NAME                        COMMENTS
             01 147-22-1111       ANTHON, CG
             02 293-88-2044       CARLYSLE, AW JR
             25 631-22-7413       WILLIAMS, ET
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                                       THE AMERICAN UNIVERSITY                              ROSTER 1023
                                          CLASS ROSTER FOR                                  PAGE 2
                                             40.280-02

                SOC SEC NBR       STUDENT'S NAME                        COMMENTS
             26 212-22-8112       WINTERS, J
             27 999-99-0123       YOUNG, PR SR
```

Edited Output Fields

10

Chapter Outline

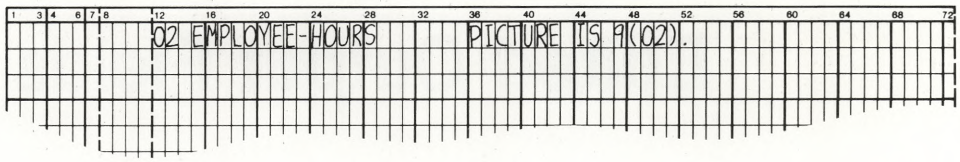


Figure 10.1
Integer variable in payroll application

Very little has been said about the manipulation of numeric fields. The COBOL report programs described thus far have made limited use of any arithmetic instructions. The ADD instruction has been used for line counters and page numbers. It is also used to accumulate the totals of certain numeric fields, such as the total of all hours worked by an employee.

All of the numeric fields that have been manipulated in these COBOL programs have been described as whole number, or integer, values. In the payroll report program, the hours an employee was credited for having worked were contained in a variable like that shown in figure 10.1.

What if an employee worked 27½ hours in a pay period? In the payroll systems that we have discussed thus far, the employee would have to be credited with having worked either 27 or 28 hours—not 27½. The first value would have shortchanged the employee for ½ hour of labor. The second value would have the employer paying for ½ hour of work not performed. Neither choice is desirable.

Defining Real Numbers in Input Records/Fields

An overwhelming number of data processing applications require computer programs to process numeric values that contain fractions of integer values. How does the COBOL programmer define a numeric field that recognizes the existence of a decimal point? An example of a variable that explicitly accounts for a decimal point is found in figure 10.2.

Implied Decimal Point
Informs the computer where a decimal point in a numeric field would be and, thus, identifies the integer and fractional portions of a numeric value.

The use of the letter *V* in a numeric field definition notes the location of an *implied decimal point*. Note that the decimal point is assumed to follow the last of the numeric digits. For that reason the two declarations of EMPLOYEE-HOURS are exactly the same. Both declarations describe integer fields. In the first instance, the absence of a decimal specification is interpreted by the COBOL compiler to mean that a decimal point does exist; however, the decimal point exists to the extreme right-hand side of the numeric field, making the field a description of an integer value. The second declaration describes the same integer field. Here, however, the programmer has made it absolutely clear that the field is an integer field.

What of the length of both fields? Does the implied decimal point specified by the letter *V* mean that the second version of EMPLOYEE-HOURS requires three positions in computer memory rather than two? The answer is no. The V represents an implied decimal point which requires no storage space whatsoever. The V simply informs the COBOL compiler that this numeric quantity is an integer. The letter *V* can also describe numeric values that do, in fact, contain fractional parts. Figure 10.3 illustrates the use of the implied decimal point to define a variable that contains *both* integer and fractional portions.

All of these declarations produce exactly the same outcome. First, a numeric field known as EMPLOYEE-WAGE is defined and allocated four positions of computer storage (remember the V isn't counted). Second, EMPLOYEE-WAGE is assumed to contain an integer portion two digits in length. Third, EMPLOYEE-WAGE is assumed to contain a fractional portion two digits in length. Consequently, the range of values that could be contained in EMPLOYEE-WAGE is 00.00 to 99.99. The decimal point is added here for clarification only.

A revised input record description illustrates the manner in which the implied decimal point is used. Also, note that the length of the numeric fields remains the same, as does that of the entire record, as shown in figure 10.4.

```
02 EMPLOYEE-HOURS          PICTURE IS 9(02).
```

and

```
02 EMPLOYEE-HOURS          PICTURE IS 9(02)V.
```

Figure 10.2
Use of implied decimal point

```
02 EMPLOYEE-WAGE       PICTURE IS 9(02)V9(02)
```

or

```
02 EMPLOYEE-WAGE       PICTURE IS 99V9(02).
```

or

```
02 EMPLOYEE-WAGE       PICTURE IS 9(02)V99.
```

or

```
02 EMPLOYEE-WAGE       PICTURE IS 99V99.
```

Figure 10.3
Alternative definitions containing integer / fractions

```
       01  TIME-CARD-INPUT.
           02 SOCIAL-SECURITY         PICTURE IS X(11).
           02 EMPLOYEE-NAME           PICTURE IS X(25).
           02 EMPLOYEE-STREET         PICTURE IS X(20).
           02 EMPLOYEE-CITY           PICTURE IS X(10).
           02 EMPLOYEE-STATE          PICTURE IS X(02).
           02 EMPLOYEE-ZIPCODE        PICTURE IS 9(05).
           02 EMPLOYEE-HOURS          PICTURE IS 9(02)V.
           02 EMPLOYEE-WAGE           PICTURE IS 99V99.
           02 FILLER                  PICTURE IS X(01).
```

Figure 10.4
Record structure with implied decimal points

Defining Real Numbers in Output Records/Fields

You should recall that the report program for the payroll application contained two record descriptions. The first description was just shown in modified form. That record describes an eighty-character input record that is not directly suitable for printing. Consequently, a second record description was developed. This record was designed to hold the data that was read from the input file. The difference, however, is that this record was designed solely for the purpose of printing. Each of the record's fields was separated by blanks to improve reading comprehension. The instruction required to reformat the data contained in the input record is shown in figure 10.5. This instruction transfers each item of data from TIME-CARD-INPUT to TIME-CARD-OUTPUT.

Our current version of TIME-CARD-OUTPUT contains numeric field definitions that have not made provision for the printing of the decimal points defined in TIME-CARD-INPUT. Consequently, the contents of EMPLOYEE-WAGE in TIME-CARD-OUTPUT would, after the MOVE CORRESPONDING instruction, be printed without a decimal point. It would be printed as an integer value.

That, unfortunately, is not the only problem. Note the two definitions that appear in figure 10.6. These definitions are equivalent. In one, the decimal point is assumed to be placed to the extreme right side of the field. In the next definition, the same location of the decimal point is explicitly stated. Both define integer fields.

Remember, these most recent definitions of EMPLOYEE-WAGE are located in TIME-CARD-OUTPUT. Data is to be transferred into this field from a field of similar name in TIME-CARD-INPUT, using a MOVE CORRESPONDING instruction. The field named EMPLOYEE-WAGE in TIME-CARD-INPUT consists of both an integer and fractional portions that are separated by an implied decimal point. The same field in TIME-CARD-OUTPUT is integer only. What will happen as a result of the MOVE CORRESPONDING instruction?

When data is moved from one numeric field to another numeric field, the numeric digits are transferred according to the decimal point location. Consider the implications of this rule in the definitions of EMPLOYEE-WAGE shown in figure 10.7. The first declaration is excerpted from TIME-CARD-INPUT; the second comes from TIME-CARD-OUTPUT.

Figure 10.5
Instruction to reformat record for printing

```
           MOVE CORRESPONDING TIME-CARD-INPUT TO
                              TIME-CARD-OUTPUT.
```

Figure 10.6
Identical definitions of integer fields

```
           02 EMPLOYEE-WAGE      PICTURE IS 9(04).
```

and

```
           02 EMPLOYEE-WAGE      PICTURE IS 9(04)V.
```

Figure 10.7
Seriously different variable definitions

```
           02 EMPLOYEE-WAGE      PICTURE IS 99V99.
```

and

```
           02 EMPLOYEE-WAGE      PICTURE IS 9(04)V.
```

The MOVE CORRESPONDING instruction would cause the following set of events to occur. First, the two digits from the integer portion of EMPLOYEE-WAGE in TIME-CARD-INPUT would be moved to the four digits allocated to EMPLOYEE-WAGE in TIME-CARD-OUTPUT. In essence, the integer portion of the first variable would be moved to the integer portion of the second variable. Recall that when the contents of smaller numeric fields are moved to larger numeric fields they are placed in the receiving field right justified. The spare positions at the extreme left are then filled with zeroes. Second, the computer attempts to transfer the fractional portion of the sending field to the fractional portion of the receiving field. Since the receiving field has no fractional portion, the fractional value being transferred is lost or truncated.

As figure 10.8 illustrates, the effect of specifying an implied decimal point in the sending field, but failing to do so in the receiving field, causes a serious misalignment of the hourly wage for each employee whose time card is processed by this report program. What started out as an hourly wage of $3.18 is interpreted by the reader that examines the printed report to be $0003.00. The lost, or truncated, eighteen cents is very important.

How is this problem corrected? At first you may be prompted to redefine EMPLOYEE-WAGE in TIME-CARD-OUTPUT to contain an implied decimal point to avoid unwanted truncation of the fractional portion of the sending field's value. Figure 10.9 shows how the redefined field would appear. This will not completely suffice, but it will cure the problem of truncation. Unfortunately, the decimal point represented by the V is only implied—no decimal point will appear in the report.

Remember, implied decimal points are not allocated computer storage in input records. Nor are implied decimal points printed as periods in output records. Thus, the numeric value associated with EMPLOYEE-WAGE in TIME-CARD-OUTPUT will be printed exactly as it was read from TIME-CARD-INPUT. Using the value just cited, the printed result is shown in figure 10.10.

Field	Picture is	Value
EMPLOYEE-WAGE OF TIME-CARD-INPUT	99V99	03 18
EMPLOYEE-WAGE OF TIME-CARD-OUTPUT	9(04)V	0003

Figure 10.8
Impact of improper decimal alignment

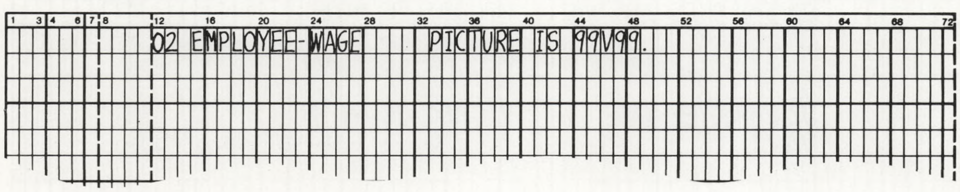
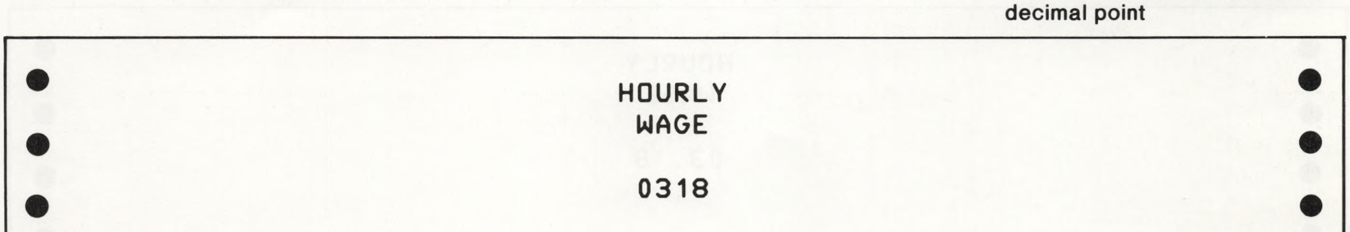


Figure 10.9
Corrected definition to avoid truncation

Figure 10.10
Truncation avoided: no printed decimal point



Printed Decimal Points

Explicit Decimal Point
Inclusion of an actual decimal point that will appear in the printed version of the numeric field that contains this editing character.

In order to print an implied decimal point, actual space in computer storage must be allocated to hold that decimal point. This involves the use of an *explicit,* rather than an *implied,* decimal point. To redefine EMPLOYEE-WAGE in TIME-CARD-OUTPUT to possess an explicit decimal point for printing, our earlier definition must be altered to look like that in figure 10.11. As you can see, the V has been replaced by an actual decimal point. Several points must be noted here. First, this field is now five digits long as opposed to the earlier four. Explicit decimal points are allocated computer storage; thus, they can be printed. Second, explicit decimal points can only be used in variables that belong to record descriptions destined to be printed using the DISPLAY instruction. Third, variables using an explicit decimal point cannot be involved in arithmetic operations. This last item will take on greater significance when arithmetic instructions are reviewed in a later chapter. Consequently, if TIME-CARD-OUTPUT is defined as shown in figure 10.12, then the sequences of MOVE CORRESPONDING and DISPLAY verbs would produce the printed output shown in figure 10.13. This printed result more closely resembles a dollar-cent value. The appearance of the decimal point does much to improve the reader's comprehension.

Figure 10.11
Field redefined with explicit decimal point

```
           02 EMPLOYEE-WAGE        PICTURE IS 99.99.
```

Figure 10.12
Output record description with explicit decimal point

```
       01  TIME-CARD-OUTPUT.
           02 FILLER               PICTURE IS X(06) VALUE SPACES.
           02 SOCIAL-SECURITY      PICTURE IS X(11).
           02 FILLER               PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-NAME        PICTURE IS X(25).
           02 FILLER               PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-STREET      PICTURE IS X(20).
           02 FILLER               PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-CITY        PICTURE IS X(10).
           02 FILLER               PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-STATE       PICTURE IS X(02).
           02 FILLER               PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-ZIPCODE     PICTURE IS 9(05).
           02 FILLER               PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-HOURS       PICTURE IS 9(02).
           02 FILLER               PICTURE IS X(04) VALUE SPACES.
           02 EMPLOYEE-WAGE        PICTURE IS 99.99.
           02 FILLER               PICTURE IS X(06) VALUE SPACES.
```

Figure 10.13
Printed result: use of decimal point

```
HOURLY
 WAGE

03.18
```

Leading Zero Suppression

If you were asked to write your age, your hourly wage, or any other number, it is unlikely that you would write it as *AGE: 0000030.* We do not include leading zeroes preceding nonzero digits in the numeric values we use so often. It stands to reason, therefore, that the leading zeroes in the previously cited hourly wage of 0318, or PAGE-001, could and should be eliminated. To avoid leading zeroes in EMPLOYEE-WAGE, the definition for that variable must be altered to look like that shown in figure 10.14. The letter *Z* indicates *zero suppression.* Z, like 9, can hold numeric digits only. Z, unlike 9, changes a leading zero into a blank. Any zero after a nonzero digit is, however, printed as a zero. The values and their printed counterparts in figure 10.15 illustrate these points.

Leading Zero Suppression
The replacement of leading zeroes in a numeric value by specified characters.

The same effect can be generated for the page numbers that report programs print as part of report titles or trailers. Examine the altered report title declaration shown in figure 10.16. This revised page number will list the two leading zeroes in the first page (001) as blanks.

```
           02 EMPLOYEE-WAGE         PICTURE IS ZZ.99.
```

Figure 10.14
Variable definition that eliminates leading zeroes

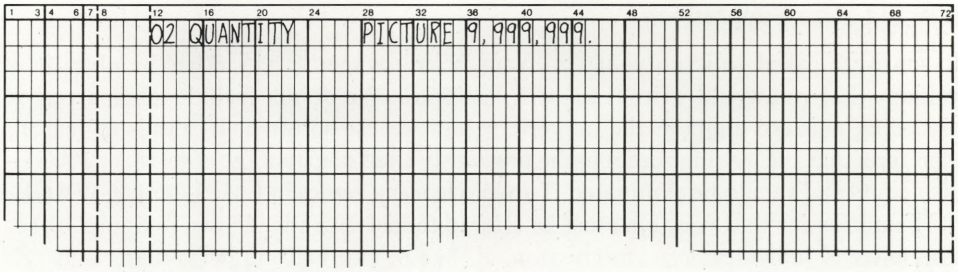
Numeric value	Printed result using ZZ.99
0018	.18
0245	2.45
1832	18.32
2000	20.00

Figure 10.15
Examples of zero suppression

```
       01  TITLE-1.
           02 FILLER              PICTURE IS X(50) VALUE SPACES.
           02 FILLER              PICTURE IS X(62) VALUE
           "ACME MACHINE COMPANY".
           02 FILLER              PICTURE IS X(05) VALUE
           "PAGE-".
           02 PAGE-NUMBER   PICTURE IS ZZ9.
```

Figure 10.16
Zero suppression for page numbers

Figure 10.17
Inserting commas for printing



Printing Commas

Often numeric values to be printed are quite large. An excellent example is the value of QUANTITY in which the numeric value on the input record spans columns 11 through 17 for a total of seven digits. A possible value might be 2365891. At first glance this value is hard to read. In fact, it is not unusual to mentally insert commas to make the value easier to pronounce. The results of this mental exercise is 2,365,891.

In order to provide the ability to print commas, our definition of QUANTITY in INVENTORY-RECORD-OUTPUT must be restructured along the lines of figure 10.17. Commas, like explicit decimal points, actually occupy computer storage. Further, when anticipating how many print positions on a printed report this variable's contents will consume, the commas must be counted as well as the 9s. In the case of QUANTITY in INVENTORY-RECORD-OUTPUT, the size of the required field is nine—seven 9s and two commas.

Zero Suppression, Commas, and Periods

If necessary the features of zero suppression, commas, and periods can be combined into a single field definition. For instance, a recent programming assignment contained a field known as TOTAL-VALUE OF INVENTORY-RECORD-OUTPUT. That field was a large numeric field that contained no provisions for commas, decimal points, and leading zero suppression. (See figure 10.18.) This original declaration not only ignored editing characters and zero suppression but it also treated the field as an integer value. Clearly, from the context of the report in which this field was to be used, the variable needs to contain integer and fractional components. Consequently, a revised declaration should contain provisions for all three items. Figure 10.18 also shows a revised definition. Rather than requiring only eight positions as the original version did, this newer version requires a total of ten positions of computer storage and print positions in the report. Finally, figure 10.18 shows the impact the edit characters have on different numeric values.

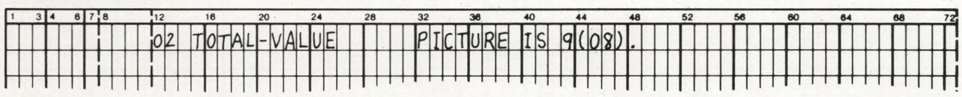
The concept of leading zero suppression is simple enough. The concept of commas is equally clear. Their combination may be somewhat less clear. The rule is that commas are printed as blanks if only leading zeroes have been encountered when transferring the numeric digits from the sending field to the receiving field. If a nonzero digit is encountered before the comma in the PICTURE IS clause, then the comma will actually be printed.

Leading Dollar Sign

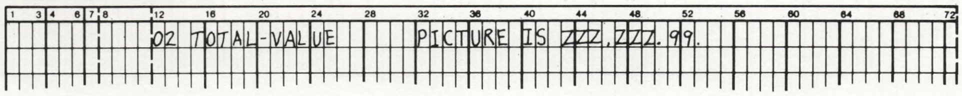
In the field just discussed, the only character missing is the dollar sign. After all, TOTAL-VALUE is a dollar-cent numeric value. If you want to include a dollar sign, then the definition for TOTAL-VALUE must be modified as shown in figure 10.19. Now the size of the field has grown to eleven, to accommodate the latest addition. The impact of the dollar sign can be seen in the examples also contained in figure 10.19. Extremely large numeric values contain no blanks between the dollar sign and the first numeric digit. Smaller numeric values provide at least one such blank and possibly more.

Figure 10.18
Definitions with and without edit characters

Original definition

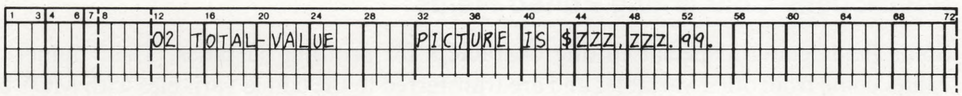


Revised definition



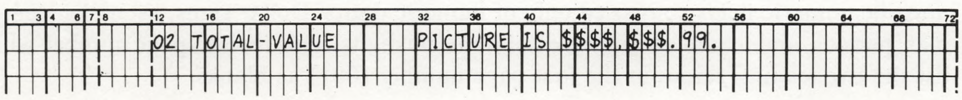
Numeric value		Printed result
00000001	➤	.01
10566720	➤	105,667.20
00034311	➤	343.11
00122228	➤	1,222.28

Figure 10.19
Impact of leading dollar sign



Numeric value		Printed result
00000001	➤	$ .01
10566720	➤	$105,667.20
00034311	➤	$ 343.11
00122228	➤	$ 1,222.28

Figure 10.20
Impact of floating dollar sign



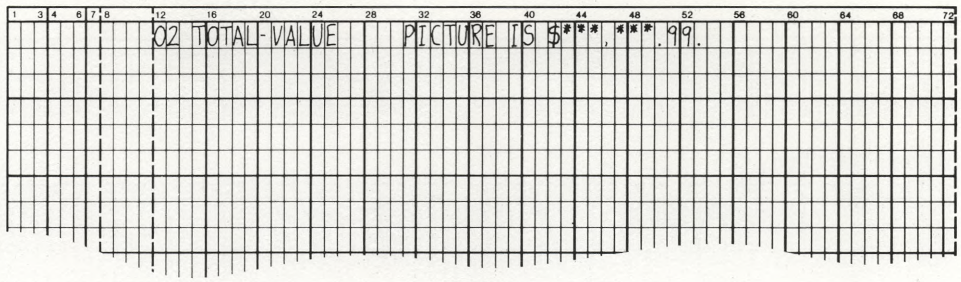
Numeric value		Printed result
00000001	➤	$.01
10566720	➤	$105,667.20
00034311	➤	$343.11
00122228	➤	$1,222.28

Floating Dollar Sign

Fixed Dollar Sign
Placing a dollar sign to the left of the first digit in a numeric field whether or not that digit is a zero.

In many check-writing applications it is not desirable to have *fixed dollar signs.* The resulting blanks can invite someone to place a check into a typewriter and imaginatively fill those blanks with numeric digits. As a means of preventing check tampering, it is possible to suppress leading zeroes with a dollar sign. The size of the field remains unchanged from figure 10.19 to figure 10.20, but the results produced are dramatically different. Note that the dollar sign is floating towards the right until a nonzero digit is encountered. In the case of $343.11, this means that the dollar sign is placed where the comma would have normally been placed. Yet you can see that this particular value is not large enough to justify the printing of the comma.

Figure 10.21
Floating asterisks



Fixed Dollar Sign, Zero Suppression by Asterisk

It is also possible to prevent the alteration of checks while allowing the dollar sign to remain fixed. In this case the leading zeroes are suppressed with a nonblank character—the asterisk. The asterisk functions as did the Z. The only difference, however, is that leading zeroes are replaced by asterisks, not blanks, as shown in figure 10.21.

Restrictions Concerning Edited Fields

Some fields contain special editing characters for printing purposes. Their numeric values come from other variables and are transferred to the edited variables with MOVE instructions. In the many examples described in this chapter, these values are supplied by record descriptions that define the layout of the input records. Because these edited variables receive their numeric values in this manner, it is inappropriate to assign an initial value to a declaration of a variable that contains any special editing characters. The declaration shown in figure 10.22 would be rejected by the COBOL compiler. The Zs are edit characters. Any variable containing an edit character in its PICTURE clause cannot also contain an initial value. Furthermore, these fields cannot be used in arithmetic operations. Remember, to be used in arithmetic operations, operands must be numeric fields composed only of numeric digits. Dollar signs, asterisks, commas, periods, and leading blanks are not numeric digits. This revised page number will list the two leading zeroes in the first page number (001) as blanks.

Printing Commas

PAGE-NUMBER has now been altered to suppress leading zeroes by replacing them with blanks. Our COBOL program contains page overflow logic that causes the value in PAGE-NUMBER to be incremented by one each time a new set of titles is printed. Unfortunately, if we attempt to increment PAGE-NUMBER directly with an ADD instruction, we encounter problems. Remember, variables that contain any editing characters cannot be involved in arithmetic operations. PAGE-NUMBER now contains an edit character—the Z.

As already stated, fields containing editing characters cannot be used in arithmetic operations. This requires that another variable be added to our program's WORKING-STORAGE SECTION. This new variable does not contain any edit characters. Instead it is strictly numeric and can be the object of an ADD instruction that can be executed every time page overflow occurs.

Figure 10.23 shows the modifications required to use both variables. Essentially, when page overflow arises, the value in the arithmetic variable—CURRENT-PAGE is incremented by one. That numeric value, in turn, is moved to PAGE-NUMBER, which contains the edit character that replaces the leading zeroes in the page number with blanks.

```
02 PAGE-NUMBER     PICTURE IS ZZ9 VALUE IS ZERO.
```

Figure 10.22
Arithmetic operations and ineligible fields

As used in the record description:

```
           01  TITLE-1.
               02 FILLER         PICTURE IS X(50) VALUE SPACES.
               02 FILLER         PICTURE IS X(62) VALUE
               "ACME MACHINE COMPANY".
               02 FILLER         PICTURE IS X(05) VALUE
               "PAGE-".
     ----->    02 PAGE-NUMBER  PICTURE IS ZZ9.
```

```
       01  CURRENT-PAGE     PICTURE IS 9(03) VALUE ZERO.
```

Figure 10.23
Use of arithmetic and edited variables

```
       01  TITLE-1.
           02 FILLER         PICTURE IS X(50) VALUE SPACES.
           02 FILLER         PICTURE IS X(62) VALUE
           "ACME MACHINE COMPANY".
           02 FILLER         PICTURE IS X(05) VALUE
           "PAGE-".
------>    02 PAGE-NUMBER  PICTURE IS ZZ9.
```

```
           ADD 1 TO CURRENT-PAGE.
           MOVE CURRENT-PAGE TO PAGE-NUMBER.
```

Defining Edited Alphanumeric Fields in Output Records

It might be concluded that only numeric fields can be edited with special characters for purposes of display. This is not entirely correct. Alphanumeric fields can be defined to contain certain editing characters as well.

The type and kind of editing characters for alphanumeric fields differ radically from those for numeric fields. Basically, the editing of alphanumeric fields is limited to partitioning those fields into smaller groups, where each group is separated by one (or more) special character(s).

For example, the input records for a sales report might contain the date on which a sale was registered. The date variable in the input record is only six characters long and is of the form MMDDYY. The report, however, desires that this date be printed with slashes that clearly partition the date into the components—month, day, and year. This can be accomplished in one of two ways. The first is much more difficult than the second.

Figure 10.24
Inserting edit characters using record structures

```
       01  SALES-RECORD-INPUT.
           .
           .
           02  DATE-OF-SALE.
               03  MONTH          PICTURE IS XX.
               03  DAY-OF-WEEK    PICTURE IS XX.
```

```
       01  SALES-RECORD-OUTPUT
           .
           .
           02  DATE-OF-SALE.
               03  MONTH          PICTURE IS XX.
               03  FILLER         PICTURE IS X VALUE "/".
               03  DAY-OF-WEEK    PICTURE IS XX.
               03  FILLER         PICTURE IS X VALUE "/".
               03  YEAR           PICTURE IS XX.
               .
```

```
           MOVE CORRESPONDING SALES-RECORD-INPUT TO
                              SALES-RECORD-OUTPUT.
```

The first alternative requires the use of record substructures and a MOVE CORRESPONDING instruction as shown in figure 10.24. A MOVE CORRESPONDING instruction transfers the components of the date and the separating slashes appear where desired.

A far simpler alternative is to make use of the special editing character—the slash. The counterpart field in the input record can then be defined not as a structure but as a single variable. The MOVE instruction automatically inserts the desired slashes. Figure 10.25 illustrates the advantages of this less cumbersome method of inserting slashes into alphanumeric fields.

Another special editing character for alphanumeric fields is the blank (B).

Summary

The input records processed by report programs contain numeric fields that may or may not possess integer values. All integer values contain implied decimal points in the forms shown in figure 10.26. In both cases the decimal point is at the extreme right-hand side of the numeric field.

These implied decimal points are not allocated any storage in computer memory. Neither are these implied decimal points actually present in the input data. When those fields are read for processing, however, the computer knows via the field declaration where a decimal point would have been had one been placed in these incoming numeric values. Printed reports that contain these numeric values should have real decimal points appearing at the proper locations. These explicit decimal points do occupy computer memory and have the effect of increasing the size of a numeric field by one position.

```
       01  SALES-RECORD-INPUT.
           .
           02 DATE-OF-SALE           PICTURE IS X(06).
           .

       01  SALES-RECORD-OUTPUT.
           .
           02 DATE-OF-SALE           PICTURE IS XX/XX/XX.
           .

           MOVE CORRESPONDING SALES-RECORD-INPUT TO
                              SALES-RECORD-OUTPUT.
```

Figure 10.25
Use of edit characters in alphanumeric field

```
           02 QUANTITY       PICTURE IS 9(06).
```

or

```
           02 QUANTITY       PICTURE IS 9(06)V.
```

Figure 10.26
Implied decimal points

Other special editing characters can be included as well. Commas can be included to improve the reader's ease of interpretation. Leading zeroes can be replaced by blanks or asterisks. Dollar signs can be added, too.

Remember that these special editing characters are added to numeric fields that are to be printed in a report. Data to be stored in a file for future processing by a computer would not make use of these special editing characters. These records should use only an implied decimal point. Also note that these special editing characters, some of which are available for alphanumeric fields, actually increase the length of the fields that make use of them. Programmers planning record descriptions for printed reports must consider this fact when accounting for each of the print positions in a 120-character line of printed output. Finally, numeric fields that contain these editing characters cannot be used in arithmetic operations. This requires the programmer to include additional field or record descriptions in the program's WORKING-STORAGE SECTION. The purpose of these new fields or records is to accumulate numeric subtotals that will later be moved to records that contain special editing characters for printing.

Questions for Review

1. If numeric data input cannot contain decimal points when processed by COBOL programs, then how does the computer know which portion of a numeric value is integer or fractional?
2. If our report title, in record structure form, contains a field named PAGE-NUMBER using any special edit character, then can PAGE-NUMBER be involved in an arithmetic operation?
3. Name and describe the alternatives for leading zero suppression.
4. Are edit characters for use in numeric values only? Explain.

Problems

1. A COBOL program contains the following variable definitions in its WORKING-STORAGE SECTION. What, if anything, will fail to operate properly in the MOVE instructions that also follow?

```
02 FIELD-1      PICTURE IS 9(4).
02 FIELD-2      PICTURE IS 9(4)V99.
02 FIELD-3      PICTURE IS 9V9(4).
02 FIELD-4      PICTURE IS V9(4).
```

a. `MOVE FIELD-1 TO FIELD-2.`
b. `MOVE FIELD-1 TO FIELD-3.`
c. `MOVE FIELD-1 TO FIELD-4.`
d. `MOVE FIELD-2 TO FIELD-3.`
e. `MOVE FIELD-2 TO FIELD-4.`
f. `MOVE FIELD-3 TO FIELD-4.`

2. Note the following input data records:

```
.........1.........2.........3.........4.........5
01.2223.5
25.5518.3
```

Examine the record structure described here as:

```
01  INPUT-RECORD.
    02 EMPLOYEE-WAGE          PICTURE IS 99V99.
    02 EMPLOYEE-HOURS         PICTURE IS 99V9.
    02 FILLER                 PICTURE IS X(73).
```

How well do the data and the record descriptions match? Why?

3. Examine the following data and corresponding editing instructions. Describe the result that would occur.

```
DATA        EDIT INSTRUCTIONS      →        RESULT
0000V87     Z(4).99
000842      ZZ9,999
```

4. What, if anything, is wrong with the following definitions?

```
Z99ZV99            .99Z
999,ZZZ          *(3).99
Z9Z9Z9         $$,$**.99
```

Programming Assignments

1. Revise the HI-FI Hut assignment from chapter 9 so that:
   a. page numbers feature leading zero suppression by blanks and
   b. all printed dollar amounts possess leading zero suppression by a floating dollar sign, commas, and decimal points.

2. Sierra Realty Systems (SRS) is a wholly owned subsidiary of Sierra Realtors and Associates of Colorado. SRS provides the automated data processing support for all eighty-seven of Sierra Realtors sales offices.

   Once each week, each sales office forwards its new home listings and sales contracts to the SRS office in Fort Collins, Colorado. These documents provide the input that serves to constitute the SRS Realty Database. One such file in this database contains a single record for each property represented and/or sold by a Sierra sales associate. The layout of each record is as follows:

Record Position	*Data Type*	*Variable*
01–03	N	Sales office number
04–05	N	Office broker
06–08	N	Sales associate
09–09	N	Property class
10–15	N	Date of listing (YYMMDD)
16–27	N	Asking price
28–40	N	Price sold
41–42	N	Commission rate
43–48	N	Date sold (YYMMDD)
49–80	AN	Property address

   To gauge the effectiveness of its sales office staffs, SRS produces an analysis of listings and sales as shown in figure 10.27. A report program to produce such a report is required.

Figure 10.27
Sierra Realty Systems report

NOTE: change in office or broker produces a new page

REPORT LAYOUT FORM

PROGRAMMER ______

PROBLEM ______

DATE ______

PAGE ____ OF ____

```
PIS-095                                    SIERRA REALTY SYSTEMS INC                                        PAGE-1,002
DATE 02-03-80                                                                                               OFFICE 103
                                           PROPERTY INVENTORY SYSTEM                                        BROKER  18

                                             LISTING/SALES REPORT

    SALES   OFFICE                  DATE  OF           PRICE
   OFFICE   BROKER  SALES            LIST              ASKED   PROPERTY
   NUMBER   NUMBER  ASSOC   CLASS    SALE               SOLD   ADDRESS

     103      18       22     R    12-20-79      $104,999.00   130 W 7TH AVE, DENVER
                                   01-05-80        96,500.00
                                                         06%

                      105     C    12-20-79      $275,500.00   HENRY BLDG, 1400 GLACIER WAY, DENVER
                                   00-00-00             0.00
                                                         07%

                      111     L    12-22-79        $3,500.00   1 ACRE, LOT 84ZP-S-10, DENVER
                                   00-00-00             0.00
                                                         10%

                  VALUE PROPERTY - SOLD          $    104,999.00
                                   UNSOLD        $    279,000.00
```

Arithmetic Instructions

11

Chapter Outline

It is difficult to imagine a COBOL program without a single arithmetic instruction. Even simple report programs that produce no subtotals have to monitor the line number currently being printed to avoid page overflow. When page overflow does appear likely, page numbers have to be generated. Consequently, COBOL programmers must learn the arithmetic subset of COBOL.

Five arithmetic instructions will be discussed in this chapter. Four are basic arithmetic instructions. A fifth, more popular, arithmetic instruction will also be introduced. The first four instructions to be examined are ADD, SUBTRACT, MULTIPLY, and DIVIDE.

ADD Instruction

Operator
The arithmetic operation specified in the COBOL statement.

Operand
The numeric variables or literals acted upon by the arithmetic operator.

The ADD instruction adds a series of arithmetic values to produce a total of those values. The values being added are known as *operands*. The arithmetic *operator*, of course, is addition. So, the operator operates (i.e., adds) upon the operands.

The list of operands whose numeric values can be added is very flexible. First, certain operands can be numeric literals. Literals can be integer only, fractional only, or possess both integer and fractional portions. Figure 11.1 illustrates several possible numeric literals. Second, the operands can also be numeric variables.

Occasionally, some beginning programmers are confused as to what constitutes a numeric variable. Simply stated, to be considered a numeric variable, that variable must have a numeric attribute (PICTURE IS 9). The confusion arises when a choice of variables is found. (See figure 11.2.) Both variables FIELD-1 and FIELD-2 contain numeric values. However, only FIELD-1 has a nine in its attribute list. While FIELD-2 might hold a numeric value, as it does in figure 11.2, it is not necessarily numeric. FIELD-2s alphanumeric attribute allows it to contain nonnumeric values as well. FIELD-1, on the other hand, can only possess a numeric value. Consequently, only FIELD-1 can be involved in an arithmetic operation.

Figure 11.1
Some alphanumeric and numeric literals

Literal	Attribute	Integer portion	Fractional portion
11.1	numeric	11	1
.42	numeric	—	42
12.	numeric	12	—
13	numeric	13	—
"1"	alphanumeric	**not numeric**	
"3.5"	alphanumeric	**not numeric**	

Figure 11.2
Numeric and alphanumeric definitions

```
       WORKING-STORAGE SECTION.

       01 FIELD-1             PICTURE IS 9(01) VALUE 5.
       01 FIELD-2             PICTURE IS X(01) VALUE "5".
```

Multiple Operands—the GIVING Option

The number of operands that may appear in a single ADD instruction is also flexible. All uses of the ADD instruction thus far have involved only two operands. In many instances, adding the values of two operands is desirable (i.e., page numbering). This, however, is not always the case.

In many circumstances, you may wish to add the values of three or more operands. If the ADD instruction permitted operation on only two operands, adding the values of three or more variables would require multiple ADD instructions and temporary storage areas.

Statement Format

```
B
↓
ADD variablename variablename [variablename...]
    GIVING variablename.
```

Figure 11.3 shows the arithmetic formula we want to carry out. Here we have a series of excerpts from a COBOL payroll report program. At some point during this program's operation, it will be necessary to compute total deductions. As a result, we must add together the values of the different tax withholdings. Assume the variables in figure 11.3 contain the appropriate tax values. With the two-operand ADD instruction, the sequence shown in figure 11.3 might seem appropriate. Unfortunately, as figure 11.3 also shows, the original values in all but one tax variable (LOCAL-WAGE-TAX) are destroyed, and this may be quite unsatisfactory.

Figure 11.3
Destroying values in two operand ADD usage

Formula

```
TOTAL DEDUCTIONS = LOCAL WAGE TAX +
                   STATE WAGE TAX +
                   FEDERAL WAGE TAX +
                   SOCIAL SECURITY TAX
```

Variable list

```
       01  EMPLOYEE-DEDUCTIONS.
           02  LOCAL-WAGE-TAX          PICTURE IS 9(08)V99.
           02  STATE-WAGE-TAX          PICTURE IS 9(08)V99.
           02  FEDERAL-WAGE-TAX        PICTURE IS 9(08)V99.
           02  SOCIAL-SECURITY-TAX     PICTURE IS 9(08)V99.
           .
           .
           02  TOTAL-DEDUCTIONS        PICTURE IS 9(10)V99.
```

Alternative one: Undesired destruction of values

```
           ADD LOCAL-WAGE-TAX TO STATE-WAGE-TAX.
           ADD STATE-WAGE-TAX TO FEDERAL-WAGE-TAX.
           ADD FEDERAL-WAGE-TAX TO SOCIAL-SECURITY-TAX.
           MOVE SOCIAL-SECURITY-TAX TO TOTAL-DEDUCTIONS.
```

Variable	Value before	Value after
LOCAL-WAGE-TAX	12 88	12 88
STATE-WAGE-TAX	53 12	66 00
FEDERAL-WAGE-TAX	241 71	307 71
SOCIAL-SECURITY-TAX	200 19	507 90
TOTAL-DEDUCTIONS	-	507 90

Figure 11.4
Preserving values by using a temporary field

Formula

```
TOTAL DEDUCTIONS = LOCAL WAGE TAX +
                   STATE WAGE TAX +
                   FEDERAL WAGE TAX +
                   SOCIAL SECURITY TAX
```

Variable list

```
 01  EMPLOYEE-DEDUCTIONS.
     02 LOCAL-WAGE-TAX          PICTURE IS 9(08)V99.
     02 STATE-WAGE-TAX          PICTURE IS 9(08)V99.
     02 FEDERAL-WAGE-TAX        PICTURE IS 9(08)V99.
     02 SOCIAL-SECURITY-TAX     PICTURE IS 9(08)V99.
     .
     .
     02 TEMPORARY-FIELD         PICTURE IS 9(10)V99.
     02 TOTAL-DEDUCTIONS        PICTURE IS 9(10)V99.
```

Alternative two: Use of temporary field to avoid destruction of values

```
     MOVE ZERO TO TEMPORARY-FIELD
     ADD LOCAL-WAGE-TAX TO TEMPORARY-FIELD.
     ADD STATE-WAGE-TAX TO TEMPORARY-FIELD.
     ADD FEDERAL-WAGE-TAX TO TEMPORARY-FIELD.
     ADD SOCIAL-SECURITY-TAX TO TEMPORARY-FIELD.
     MOVE TEMPORARY-FIELD TO TOTAL-DEDUCTIONS.
```

Variable	Value before	Value after
LOCAL-WAGE-TAX	12 88	12 88
STATE-WAGE-TAX	53 12	53 12
FEDERAL-WAGE-TAX	241 71	241 71
SOCIAL-SECURITY-TAX	200 19	200 19
TEMPORARY-FIELD	-	507 90
TOTAL-DEDUCTIONS	-	507 90

Temporary Fields
Variables in the DATA DIVISION required to hold the intermediate products of arithmetic formulas. Temporary fields are required because, when separate arithmetic verbs are used, a formula must be executed in parts, and the results of each part must be saved without destroying the contents of the original variables.

To avoid destroying the values of the variables involved, figure 11.4 contains a new field. The use of TEMPORARY-FIELD permits the tax withholding variables to retain their original values. However, this particular alternative requires more ADD instructions than the number that was originally used in figure 11.3.

A third alternative, preferred over those shown in figures 11.3 and 11.4 can be found in figure 11.5. The GIVING option of the ADD instruction has two advantages. First, fewer variables are needed and TEMPORARY-FIELD is not necessary. Second, fewer ADD instructions are required to add all of the tax withholding values.

One common point of confusion among beginning programmers is whether the TO and GIVING options can be combined in a single ADD instruction. They cannot. The TO and GIVING options are mutually exclusive. You may use either the TO option or the GIVING option, but not both.

Multiple Operands—the TO Option

One disadvantage of the GIVING option is that the value of the receiving field is replaced by the sum of the values being added together. So, the variable TOTAL-DEDUCTIONS in figure 11.4 loses whatever value it held prior to the ADD instruction. This is not always desirable.

Figure 11.5
Use of GIVING option

Formula

```
TOTAL DEDUCTIONS = LOCAL WAGE TAX +
                   STATE WAGE TAX +
                   FEDERAL WAGE TAX +
                   SOCIAL SECURITY TAX
```

Variable list

```
       01  EMPLOYEE-DEDUCTIONS.
           02  LOCAL-WAGE-TAX            PICTURE IS 9(08)V99.
           02  STATE-WAGE-TAX            PICTURE IS 9(08)V99.
           02  FEDERAL-WAGE-TAX          PICTURE IS 9(08)V99.
           02  SOCIAL-SECURITY-TAX       PICTURE IS 9(08)V99.
           .
           .
           02  TOTAL-DEDUCTIONS          PICTURE IS 9(10)V99.
```

Alternative three: Use of GIVING option

```
           ADD LOCAL-WAGE-TAX,
               STATE-WAGE-TAX,
               FEDERAL-WAGE-TAX,
               SOCIAL-SECURITY-TAX,
                 GIVING
                     TOTAL-DEDUCTIONS.
```

Variable	Value before	Value after
LOCAL-WAGE-TAX	12 88	12 88
STATE-WAGE-TAX	53 12	53 12
FEDERAL-WAGE-TAX	241 71	241 71
SOCIAL-SECURITY-TAX	200 19	200 19
TOTAL-DEDUCTIONS	-	507 90

To return to our hypothetical payroll program, deductions usually involve more than withholding taxes, and may include insurances, credit union, deductions, and so on. Thus, TOTAL-DEDUCTIONS does not contain total deductions until these other deductions are included.

Figure 11.6 shows the complete list of deductions for our hypothetical firm. Remember, at this point TOTAL-DEDUCTIONS contains the total of withholding taxes only. We also want to add the other deductions to TOTAL-DEDUCTIONS. This can be accomplished, as alternative one suggests, by a separate ADD instruction for each nontax variable. However, the entire process can be completed with a single ADD instruction by using the now familiar TO option.

Statement Format

```
B
↓
ADD variablename variablename [variablename...]
    TO variablename.
```

The TO option, unlike the GIVING option, does not ignore the values in the receiving field. By using the TO option, TOTAL-DEDUCTIONS does not forget the total of withheld taxes. Instead, those tax withholdings are added to the remaining nontax deductions to finally produce a total of all deductions.

Figure 11.6
Use of multiple operands using TO option

Formula

```
TOTAL DEDUCTIONS = LOCAL WAGE TAX +
                   STATE WAGE TAX +
                   FEDERAL WAGE TAX +
                   SOCIAL SECURITY TAX +
                   HEALTH INSURANCE +
                   LIFE INSURANCE +
                   CREDIT UNION
```

Variable list

```
       01  EMPLOYEE-DEDUCTIONS.
           02  LOCAL-WAGE-TAX           PICTURE IS 9(08)V99.
           02  STATE-WAGE-TAX           PICTURE IS 9(08)V99.
           02  FEDERAL-WAGE-TAX         PICTURE IS 9(08)V99.
           02  SOCIAL-SECURITY-TAX      PICTURE IS 9(08)V99.
           02  HEALTH-INSURANCE         PICTURE IS 9(04)V99.
           02  LIFE-INSURANCE           PICTURE IS 9(04)V99.
           02  CREDIT-UNION             PICTURE IS 9(04)V99.
           02  TOTAL-DEDUCTIONS         PICTURE IS 9(10)V99.
```

Alternative one: Adding other values with separate ADDs

```
           ADD HEALTH-INSURANCE TO TOTAL-DEDUCTIONS.
           ADD LIFE-INSURANCE TO TOTAL-DEDUCTIONS.
           ADD CREDIT-UNION TO TOTAL-DEDUCTIONS.
```

Alternative two: Multiple operands

```
           ADD HEALTH-INSURANCE,
               LIFE-INSURANCE,
               CREDIT-UNION
             TO
               TOTAL-DEDUCTIONS.
```

Multiple Receiving Fields—Both TO and GIVING Options

At this point, it might appear that the only location of multiple operands is the sending portion of the ADD instruction. Figures 11.4 and 11.5 show multiple operands as being those variables whose values are to be added together. Multiple-receiving operands are also possible—whether multiple-sending operands are used or not.

When multiple-receiving operands are specified using the GIVING option, the arithmetic sum produced by the ADD instruction replaces whatever values those receiving operands held before the instruction was executed. Using the TO option in similar circumstances does not produce the same results.

Statement Format

```
B
↓
ADD variablelist {GIVING} variable list.
                 {TO    }
```

The TO option causes the value previously held in the receiving operand to be combined with the other numeric values in the addition operation. Consequently, the values in the respective receiving operands are not totally lost, as was the case with the GIVING option. They are reflected in the final values of those variables.

Figure 11.7
Use of CORRESPONDING option for group operations

```
       01  DEPARTMENT-TOTALS.
           02  DEPARTMENT-IDENTIFICATION   PICTURE IS X(03).
           02  GOODS-SOLD                  PICTURE IS 9(10)V99.
           02  TAXES-COLLECTED             PICTURE IS 9(08)V99.
           02  UNITS-SOLD                  PICTURE IS 9(10).
       01  DIVISION-TOTALS.
           02  DIVISION-IDENTIFICATION     PICTURE IS X(02).
           02  GOODS-SOLD                  PICTURE IS 9(12)V99.
           02  TAXES-COLLECTED             PICTURE IS 9(10)V99.
           02  UNITS-SOLD                  PICTURE IS 9(12).
```

Alternative one

```
       DEPARTMENT-CONTROL-BREAK SECTION.
       D-C-B-001.
           ADD GOODS-SOLD OF DEPARTMENT-TOTALS TO
               GOODS-SOLD OF DIVISION-TOTALS.
           ADD TAXES-COLLECTED OF DEPARTMENT-TOTALS TO
               TAXES-COLLECTED OF DIVISION-TOTALS.
           ADD UNITS-SOLD OF DEPARTMENT-TOTALS TO
               UNITS-SOLD OF DIVISION-TOTALS.
           MOVE ZEROES TO GOODS-SOLD OF DEPARTMENT-TOTALS.
           .
           .
```

Alternative two

```
       DEPARTMENT-CONTROL-BREAK SECTION.
       D-C-B-001.
           ADD CORRESPONDING DEPARTMENT-TOTALS TO
                                 DIVISION-TOTALS.
           MOVE ZEROES TO GOODS-SOLD OF DEPARTMENT-TOTALS.
           .
           .
```

Group ADD Instructions

Another option of the ADD instruction is not well known. This particular option allows multiple two-operand addition operations in a single ADD instruction.

Recall from our control break programs that when a change in a key field occurs, we may want to add—for instance—the values for departments to like values for divisions. Examine the contents of figure 11.7. Alternative one shows that each of the variables would require its own ADD instruction. Yet since both record structures contain identically named variables, the CORRESPONDING option may be used. Another advantage of the CORRESPONDING option is that the addition of new, identically named variables requires no change to the ADD instruction. Alternative two illustrates this most recent option of the ADD instruction. If the CORRESPONDING option had not been used, each new variable would have required an additional ADD instruction.

Statement Format

```
B
↓
ADD CORRESPONDING recordname TO recordname.
```

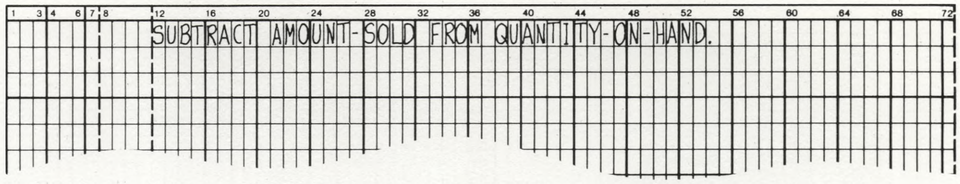


Figure 11.8
Format of SUBTRACT instruction

SUBTRACT Instruction

From time to time COBOL programs must be capable of subtracting numeric values from other numeric values. This is accomplished through the SUBTRACT verb. For instance, in figure 11.8 we desire to update the value in QUANTITY-ON-HAND after selling an unspecified number of items of a particular item from our inventory.

In the SUBTRACT instruction, we deduct the value represented by the first operand from the value represented by the second operand. As in the ADD instruction, the result of this arithmetic operation is placed in the second operand on conclusion of the instruction. Further, the SUBTRACT instruction (like ADD) permits the usage of multiple operands.

Statement Format

```
B
↓
SUBTRACT variablename FROM variablename.
```

Statement Format

```
B
↓
SUBTRACT variablelist FROM variablename.
```

If you will now return to our payroll report program you will notice that it is possible to compute an individual's net pay only if gross pay and all deductions are known. Examine the COBOL program excerpts in figure 11.9. Note that all tax and insurance withholdings are effectively combined to produce a net pay value.

Unfortunately, the method of computing net pay shown in alternative one destroys the original value held in GROSS-PAY. The original value of GROSS-PAY may still be required. Consequently, the SUBTRACT instruction (like ADD) permits the use of the GIVING option, as shown in alternative two. Now the original value of GROSS-PAY is not lost as a result of the payroll computation.

Statement Format

```
B
↓
SUBTRACT variablelist FROM variablename
     GIVING variablelist.
```

Figure 11.9 also illustrates that the SUBTRACT statement can make use of multiple operands. While it is not shown here, multiple-receiving operands can also be specified. The result produced is precisely the same as when multiple-receiving fields were used in the GIVING option of the ADD instruction.

Statement Format

```
B
↓
SUBTRACT CORRESPONDING recordname FROM recordname.
```

Formula

```
TOTAL DEDUCTIONS = LOCAL WAGE TAX +
                   STATE WAGE TAX +
                   FEDERAL WAGE TAX +
                   SOCIAL SECURITY TAX +
                   HEALTH INSURANCE +
                   LIFE INSURANCE +
                   CREDIT UNION
```

Figure 11.9
Certain options in SUBTRACT instruction

Variable list

```
       01  EMPLOYEE-DEDUCTIONS.
           02  LOCAL-WAGE-TAX              PICTURE IS 9(08)V99.
           02  STATE-WAGE-TAX              PICTURE IS 9(08)V99.
           02  FEDERAL-WAGE-TAX            PICTURE IS 9(08)V99.
           02  SOCIAL-SECURITY-TAX         PICTURE IS 9(08)V99.
           02  HEALTH-INSURANCE            PICTURE IS 9(04)V99.
           02  LIFE-INSURANCE              PICTURE IS 9(04)V99.
           02  CREDIT-UNION                PICTURE IS 9(04)V99.
           02  TOTAL-DEDUCTIONS            PICTURE IS 9(10)V99.
       01  GROSS-PAY                       PICTURE IS 9(12)V99.
       01  NET-PAY                         PICTURE IS 9(12)V99.
```

Alternative one: Destroying value with SUBTRACT

```
           SUBTRACT LOCAL-WAGE-TAX,
                    STATE-WAGE-TAX,
                    FEDERAL-WAGE-TAX,
                    SOCIAL-SECURITY-TAX,
                    HEALTH-INSURANCE,
                    LIFE-INSURANCE,
                    CREDIT-UNION
               FROM
                    GROSS-PAY.
```

Alternative two: Preserving values with GIVING option

```
           SUBTRACT LOCAL-WAGE-TAX,
                    STATE-WAGE-TAX,
                    FEDERAL-WAGE-TAX,
                    SOCIAL-SECURITY-TAX,
                    HEALTH-INSURANCE,
                    LIFE-INSURANCE,
                    CREDIT-UNION
               FROM
                    GROSS-PAY
               GIVING
                    NET-PAY.
```

MULTIPLY Instruction

The third arithmetic operator is multiplication. As was the case for both ADD and SUBTRACT, MULTIPLY has more than one format. Both possible formats are illustrated in figure 11.10.

Statement Format

```
B
↓
MULTIPLY variablename BY variablename.
```

Figure 11.10
Alternative formats of the MULTIPLY instruction

```
       01  TIME-CARD-INPUT.
           .
           02 HOURS-WORKED        PICTURE IS 9(02).
           02 HOURLY-WAGE         PICTURE IS 9(02)V99.
           .
       01  EMPLOYEE-DEDUCTIONS.
           .
           .
       01  GROSS-PAY              PICTURE IS 9(12)V99.
```

Alternative one: Value destroyed and possible truncation occurs

```
           MULTIPLY HOURS-WORKED BY HOURLY-WAGE.
```

Alternative two: Value preserved and no truncation occurs

```
           MULTIPLY HOURS-WORKED BY HOURLY-WAGE
                GIVING
                   GROSS-PAY.
```

Figure 11.10 illustrates the two ways to complete the multiplication operation. The first alternative would cause the product of the operation, gross pay, to be placed in the variable HOURLY-WAGE. However, an examination of HOURLY-WAGE shows that the field is just barely large enough to contain a wage—it is not large enough to hold a gross pay value without truncation occurring. Consequently, the second alternative is more often used. Now both HOURS-WORKED and HOURLY-WAGE retain their original values, and truncation of the gross pay value does not occur.

Statement Format

```
B
↓
MULTIPLY variablename BY variablename
   GIVING variablename.
```

DIVIDE Instruction

The last of the individual arithmetic instructions permits the COBOL programmer to divide numeric values into the contents of numeric fields. The two, more popular forms of this particular instruction are shown in figure 11.11. These two DIVIDE instructions could easily be added to the COBOL report program used in our earlier examples. In this context, it might be highly desirable to compute the average commission paid to a salesperson in each district, each region, and the entire company. This would require the record descriptions named DISTRICT-SUBTOTAL and DISTRICT-SUMMARY-LINE to be modified, or to contain the appropriate numeric subtotals and edited output fields, respectively.

Statement Format

```
B
↓
DIVIDE variablename BY variablename.
```

```
02 NUMBER-OF-SALESPERSONS      PICTURE IS 999V.
```

Alternative one:

```
DIVIDE COMMISSION OF DISTRICT-SUBTOTAL
        BY
                NUMBER-OF-SALESPERSONS.
```

Alternative two:

```
DIVIDE COMMISSION OF DISTRICT-SUBTOTAL
        BY
                NUMBER-OF-SALESPERSONS
        GIVING
                AVERAGE-COMMISSION OF DISTRICT-SUMMARY-LINE.
```

Figure 11.11
Alternative formats of the DIVIDE instruction

Figure 11.12
More complex mathematic formula

```
NETPAY = (HOURS * WAGE) - ( (FED-TAX-RATE * HOURS * WAGE) +
                            (STATE-TAX-RATE * HOURS * WAGE) +
                            (RETIREMENT-RATE * HOURS * WAGE) )
```

It should be noted, however, that the first DIVIDE instruction would place the result of the operation into a numeric field in the DATA DIVISION known as NUMBER-OF-SALESPERSONS. This numeric field might be defined as shown in figure 11.11. The intent of the DIVIDE instruction is to compute an average commission, whether it be for the district, the region, or the whole company. Commission is a dollar-cent value and does not correspond to the attributes defined for NUMBER-OF-SALESPERSONS. In fact, if the first form of the DIVIDE instruction were used, what was intended to be a dollar-cent value would be forced to fit into a numeric field that was intended to accommodate an entirely different type of numeric entity. Truncation of the leading dollar portion of such a statistic is also possible if forced into this field. The cent value would be lost entirely because NUMBER-OF-SALESPERSONS is an integer value. For these reasons, the second form of the DIVIDE instruction is preferred.

Statement Format

```
B
↓
DIVIDE variablename BY variablename.
     GIVING variablename.
```

COMPUTE Instruction

The individual arithmetic instructions are sufficient for COBOL programs that require very little computation. In many instances, however, COBOL programs involve so much computation that use of these separate arithmetic instructions would be too cumbersome. For instance, examine the payroll formula shown in figure 11.12. To convert this formula into COBOL by using ADD, SUBTRACT, and MULTIPLY instructions could be quite tedious. Many separate COBOL statements would be necessary.

Figure 11.13
Examples of the COMPUTE instruction

COMPUTE GROSS-PAY — HOURS-WORKED * HOURLY-WAGE.

Addition	ADD 1 TO CURRENT-LINE. ADD STATE TO FEDERAL GIVING DEDUCTIONS.	COMPUTE CURRENT - LINE = CURRENT-LINE + 1. COMPUTE DEDUCTIONS = STATE + FEDERAL.
Subtraction	SUBTRACT DEDUCTIONS FROM GROSS-PAY. SUBTRACT DEDUCTIONS FROM GROSS-PAY GIVING NET-PAY.	COMPUTE GROSS-PAY = GROSS-PAY – DEDUCTIONS. COMPUTE NET-PAY = GROSS-PAY – DEDUCTIONS.
Multiplication	MULTIPLY HOURS-WORKED BY HOURLY-WAGE. MULTIPLY HOURS-WORKED BY HOURLY-WAGE GIVING SALARY-EARNED.	COMPUTE HOURLY-WAGE= HOURS-WORKED* HOURLY-WAGE. COMPUTE SALARY-EARNED = HOURLY-WAGE * HOURS-WORKED.
Division	DIVIDE NBR-OF-PERSONS INTO COMMISSION. DIVIDE NBR-OF-PERSONS INTO COMMISSION GIVING AVE-COMMISSION.	COMPUTE COMMISSION= COMMISSION/ NBR-OF-PERSONS. COMPUTE AVE-COMMISSION = COMMISSION/NBR-OF-PERSONS.

To minimize the number of COBOL statements required, the COMPUTE verb was included as a part of COBOL. Sample formats of this instruction are shown in figure 11.13. To execute the first instruction, the numeric value in HOURS-WORKED is multiplied by the numeric value in HOURLY-WAGE. Then the result is placed in a numeric variable known as GROSS-PAY. Thus, this usage of the COMPUTE verb is much like the MULTIPLY verb with the GIVING option. Indeed, each of the single arithmetic verbs has a counterpart in the COMPUTE verb, as shown in figure 11.13. The five arithmetic operators for the COMPUTE verb are:

1. plus sign (+) for addition,
2. minus sign (−) for subtraction,
3. asterisk (*) for multiplication,
4. slash (/) for division, and
5. double asterisk (**) for exponentiation

Statement Format

```
B
↓
COMPUTE variablename = expression.
```

```
       COMPUTE NET-PAY = HOURLY-WAGE *
                             HOURS-WORKED -
                         HOURLY-WAGE *
                             HOURS-WORKED * FED-TAX-RATE -
                         HOURLY-WAGE *
                             HOURS-WORKED * STATE-TAX-RATE -
                         HOURLY-WAGE *
                             HOURS-WORKED * RETIREMENT-RATE.
```

or

```
       COMPUTE GROSS-PAY = HOURLY-WAGE * HOURS-WORKED.
       COMPUTE NET-PAY = GROSS-PAY -
                         GROSS-PAY * FED-TAX-RATE -
                         GROSS-PAY * STATE-TAX-RATE -
                         GROSS-PAY * RETIREMENT-RATE.
```

or

```
       MULTIPLY HOURLY-WAGE BY HOURS-WORKED
                GIVING
                       GROSS-PAY.
       MULTIPLY FED-TAX-RATE BY GROSS-PAY
                GIVING
                       FED-TAX-DEDUCTION.
       MULTIPLY STATE-TAX-RATE BY GROSS-PAY
                GIVING
                       STATE-TAX-DEDUCTION.
       MULTIPLY RETIREMENT-RATE BY GROSS-PAY
                GIVING
                       RETIREMENT-DEDUCTION.
       ADD FED-TAX-DEDUCTION STATE-TAX-DEDUCTION
                GIVING
                       TOTAL-DEDUCTIONS.
       ADD RETIREMENT-DEDUCTION TO TOTAL-DEDUCTIONS.
       SUBTRACT TOTAL-DEDUCTIONS FROM GROSS-PAY
                GIVING
                       NET-PAY.
```

Figure 11.14
COMPUTE versus other arithmetic verbs

As mentioned earlier, the power in the COMPUTE verb lies in the programmer's ability to combine arithmetic operations into a single COBOL statement that would otherwise require many separate statements. The payroll formula mentioned earlier serves as an excellent example. Figure 11.14 illustrates—for comparative purposes—two uses of the COMPUTE verb in the NET-PAY computation as well as a string of single-function arithmetic COBOL instructions. The brevity offered by the COMPUTE statement often makes it a more favored verb than combinations of ADD, SUBTRACT, MULTIPLY, and DIVIDE.

```
COMPUTE D = A - B * C.
```

Figure 11.15
Order of processing arithmetic operators

Order of Processing Operators

The last example of the COMPUTE verb for NET-PAY computation shows both multiplication and subtraction operators in the same statement. When more than one type of arithmetic operator is specified, the programmer must exercise enough care to enable the COBOL compiler to interpret the arithmetic expression as the programmer had intended. Examine the COMPUTE instruction shown in figure 11.15. If the programmer wants the numeric value in B to be multiplied by C and the product of that operation to be subtracted from A, this example is correct as shown. On the other hand, if the programmer really wants B to be subtracted from A and the result multiplied by C, the programmer will not receive the desired outcome.

Operator Precedence
Determining which arithmetic operators are to act in what sequence, based upon the priorities of those arithmetic operators.

In COBOL, the arithmetic operators are assigned priorities. Operators with higher priorities are carried out before operators with lower priorities. Of the five operators discussed, exponentiation has the highest priority, multiplication and division have the next highest priority; addition and subtraction have a lower priority. Consequently, multiplication and division operators will be executed prior to addition and subtraction operators. This is why B is multiplied by C before the requested subtraction operation—multiplication has a higher priority than subtraction.

Changing Processing Order

What if the programmer wants the subtraction operation to be executed before the multiplication operation? This requires the programmer to write two COMPUTE statements. Figure 11.6 illustrates two approaches that can bring about the desired outcome. Unfortunately, this solution causes two statements to come into existence where before there had been only one, and it also requires the definition of another variable in the program's WORKING-STORAGE SECTION: TEMPORARY. For these reasons another solution is offered.

Parentheses
A grouping device that specifies which part of arithmetic formulas are to be executed first, second, third, etc.

COBOL permits a programmer to isolate lower priority portions of arithmetic expressions with *parentheses.* A subexpression (i.e., part of an expression) enclosed in parentheses will be executed before the part of the expression located outside of the parentheses. Our earlier COMPUTE statement can be restructured to appear as the second or third alternatives shown in figure 11.16. Now B is subtracted from A prior to the multiplication operation.

More than one set of parentheses can appear in a single arithmetic expression. In fact, many such parenthesized subexpressions can occur. The third alternative in figure 11.16 illustrates the use of parenthesized subexpressions. The subtraction operation will still take place prior to the multiplication operation.

When nested parentheses are used (i.e., one set inside another), you must be sure that the number of left parentheses are equalled (balanced) by the number of right parentheses.

```
       COMPUTE TEMPORARY = A - B.
       COMPUTE D = TEMPORARY * C.
```

or

```
       COMPUTE D = (A - B) * C.
```

or

```
       COMPUTE D = ((A - B) * C).
```

Figure 11.16
Forcing the order of interpretation

```
       02 QUANTITY-ONHAND      PICTURE IS 9(05)V
                               USAGE IS DISPLAY.
```

Figure 11.17
USAGE IS DISPLAY in variable definition

Negative Values

At the beginning of this text all numeric values were integer only. Then with the introduction of edited field attributes, noninteger values (i.e., real numbers) were also introduced. However, all of these numeric values were positive. Can COBOL accommodate negative numbers? The answer is a very guarded yes.

All of the numeric field definitions can actually be written as shown in figure 11.17. This variable definition contains a new phrase—USAGE IS DISPLAY. What does this phrase mean?

USAGE IS DISPLAY
A default attribute for all COBOL variables that specifies that the data in each variable are stored in human-readable form. Such data require conversion if it is to be involved in any arithmetic operation.

All numeric variables can be one of two types in COBOL. A numeric variable can be represented in computer storage in external (DISPLAY) form or in internal form, as specified by the programmer. DISPLAY form data are stored exactly as they were read from an input record. That is, DISPLAY data are in human-readable form. Computers do not manipulate data in human-readable form. Instead, data must be converted to an internal form and then manipulated. In a sense, externally stored variables are more difficult to operate on than internally stored variables. First, external variables must be converted to an internal representation. Second, the variable can then be manipulated by the computer, perhaps by using an ADD instruction. If the value of the variable is modified, the new value must be transferred from its internal form to its external form and placed back in the variable from which it came.

Statement Format

```
B
↓
levelnumber variablename PICTURE IS attributes
                         [USAGE IS DISPLAY]
                         [VALUE literal].
```

Internally formatted variables can easily be defined to accommodate an indication of sign. Numeric values can readily be tagged and manipulated as positive or negative entities. This is not true with externally formatted variables. Remember, all the variables presented thus far have been external. Internal variables will be introduced and discussed in greater detail in a later chapter.

Figure 11.18
Adding signs to numeric variables and values

```
       02 QUANTITY-SOLD     PICTURE IS S9(05)V.
```

External Variables with Signs

An external variable can be defined with an algebraic sign as shown in figure 11.18. As you can see, a sign is indicated by the letter *S*. The S must precede the 9, which indicates the field is numeric. The phrase USAGE IS DISPLAY need not be provided. It is assumed to be present.

Signed, External Fields on Input

What if a numeric value to be read from an input file as part of a record is negative rather than positive? How is this done? Perhaps the answer will indicate why negative numbers present a less than ideal set of circumstances in USAGE IS DISPLAY variables.

Assume an input record from our inventory report program contains the record description shown in figure 11.19. Further, assume that the value corresponding to this particular variable can be found in columns 20 through 24. Examples of typical record images are also shown in figure 11.19. Each of the values shown is positive and would be read and processed as such. How would you make the value negative in the second record, for instance? The answer is to concentrate on the least significant digit.

Externally formatted data can be tagged as negative by altering the value of the extreme right-hand digit within the specific field. Since this digit must be numeric (the field that contains the digit is numeric), it is represented in EBCDIC,[1] as shown in figure 11.20. Notice how each digit, in EBCDIC form, has an F in its extreme left-hand nibble?[2]

To convert each numeric digit to its negative counterpart, the letter *F* must be changed to the letter *D*. This would produce the conversion chart shown in figure 11.21.

This conversion chart indicates that to change the value of QUANTITY-SOLD in the second record, the number must appear in that record not as 1014 (positive), but as 101M instead. This means that every negative number will have a letter of the alphabet as its least significant digit.

Signed, External Fields on Output

The impact that negative numbers have on printed reports depends on what type of output field definition is used. Remember, numeric values have two types of output field definitions. For comparative purposes they are fields that contain editing characters and those that do not.

When a negative number is moved into a numeric field that does not also contain edit characters, then the least significant digit prints as either a blank or a letter of the alphabet. Thus, if −1014 were moved to such a field and then printed, the value appearing on the printed report would be 101M. This would prove very difficult to interpret.

If, however, a field definition contains any special editing characters, the negative number can be handled so that the letters of the alphabet will not print. All of the numeric digits will print; but, unfortunately, the negative sign will not.

Several ways to note that a numeric value is negative are shown in figure 11.22. In each case a special value is appended to the numeric field when it is printed to indicate that the number is negative. For example, in the first instance a trailing negative sign will appear, in the second instance a CR will appear, and in the third instance a DB will appear. For any of these suffixes to appear, the value must be negative. Otherwise, the suffix will be replaced by a blank or blanks.

```
01  INVENTORY-INPUT-RECORD.
    .
    .
    02  QUANTITY-SOLD         PICTURE IS S9(05)V.
    .
    .
    .
```

```
. . . . . .  1  . . . . . .  2  . . . . . .  3  . . . . . .  4  . . . . . .
RECORD 1 ——→         0084
RECORD 2 ——→         1014
RECORD 3 ——→         0001
    .
    .
    .
RECORD N ——→         7624
```

Figure 11.19
Describing numeric data by record description

Digit in display form	Representation in EBCDIC
0	F0
1	F1
2	F2
3	F3
4	F4
5	F5
6	F6
7	F7
8	F8
9	F9

Figure 11.20
EBCDIC representation of numeric digits

Display positive	EBCDIC positive		EBCDIC negative	Display negative
0	F0	——→	D0	BLANK*
1	F1	——→	D1	J
2	F2	——→	D2	K
3	F3	——→	D3	L
4	F4	——→	D4	M
5	F5	——→	D5	N
6	F6	——→	D6	O
7	F7	——→	D7	P
8	F8	——→	D8	Q
9	F9	——→	D9	R

*no printed character

Figure 11.21
Transforming positive values to negative values

```
    02  QUANTITY-SOLD         PICTURE IS Z,ZZ9-.
```

or

```
    02  QUANTITY-SOLD         PICTURE IS Z,ZZ9CR.
```

or

```
    02  QUANTITY-SOLD         PICTURE IS Z,ZZ9DB.
```

Figure 11.22
Edited fields and negative values

Proper Field Size

One of the greatest problems in numeric field manipulation is determining the proper size of the numeric fields. Note the COBOL declarations and the arithmetic instruction shown in figure 11.23. When HOURS-WORKED is multiplied by HOURLY-WAGE, it is possible—given the size of each field—to produce a salary as large as, for instance, 989.01. This maximum salary is one digit larger than the number of digits in the definition of SALARY-EARNED. Consequently, truncation of the most significant digit would occur—the extreme left-hand digit would be lost. Unintended truncation in any field, numeric or not, is almost always an undesirable outcome. In the example just noted, someone expecting a paycheck of 989.01 will instead receive a check for 89.01.

When truncation of any digit within a numeric field occurs, the computer continues to process the program with no warning that truncation has taken place. Only two alternatives are possible to avoid truncation of leading digits in numeric fields. First, when defining the receiving fields—fields that will hold the outcome of arithmetic operations—make sure they can accommodate the largest value produced as a result of an arithmetic operation. Had this philosophy been followed in the example, SALARY-EARNED would have had a PICTURE IS clause of 999V99.

Even though the multiplication operation noted earlier could only produce a maximum value of 989.01, this new definition can hold a number slightly larger. In any event, truncation would not occur now unless the definitions for HOURLY-WAGE and/or HOURS-WORKED were altered.

A second, less often used, alternative would be to include an option in each of the arithmetic verbs. The phrase ON SIZE ERROR, shown in figure 11.24, can be a part of each arithmetic verb. This newest portion of an arithmetic instruction can only become active if truncation of a leading numeric digit takes place. If truncation should occur in the example listed here, control would be turned over to a section in our PROCEDURE DIVISION named SIZE-ERROR-ROUTINE. SIZE-ERROR-ROUTINE would probably contain a series of COBOL statements to print a truncation warning message on the report being produced. This would allow the reader of the report to contact the programmer to redefine the attributes of the fields involved, and thereby eliminate the problem.

Figure 11.23
Problems with numeric field size

```
02 HOURS-WORKED        PICTURE IS 99V.
02 HOURLY-WAGE         PICTURE IS 9V99.
02 SALARY-EARNED       PICTURE IS 99V99.

MULTIPLY HOURS-WORKED BY HOURLY-WAGE
      GIVING
            SALARY-EARNED.
```

Figure 11.24
Use of ON SIZE ERROR option

```
MULTIPLY HOURS-WORKED BY HOURLY-WAGE
      GIVING
            SALARY-EARNED
      ON SIZE ERROR
            PERFORM SIZE-ERROR-ROUTINE.
```

Statement Format

```
B
↓
⎧ADD     ⎫
⎪SUBTRACT⎪   operands/directives ON SIZE ERROR statement(s)
⎨MULTIPLY⎬
⎪DIVIDE  ⎪
⎩COMPUTE ⎭
```

As mentioned earlier, however, the ON SIZE ERROR option is seldom used by COBOL programmers. It is more reasonable to define numeric fields large enough to hold whatever values a program attempts to store.

Summary

COBOL presents a total of five arithmetic verbs for possible use. In general, these five alternatives can be partitioned into two distinct groups. The first group features a different verb for each of the basic arithmetic operations—ADD, SUBTRACT, MULTIPLY, and DIVIDE. The second group contains only the COMPUTE verb. However, this one verb permits the programmer to perform each of the four separate arithmetic operations, and to combine these operations in a single COBOL statement. Using the basic arithmetic operations, anything other than very simple formulas may require many ADD, SUBTRACT, MULTIPLY, and DIVIDE instructions—possibly with temporary, intermediate product variables in the DATA DIVISION. The COMPUTE instruction, on the other hand, can represent complex formulas without forcing the programmer to define temporary fields—and it can do so in a single statement.

It should be noted that the COMPUTE verb is, relatively speaking, a recent addition to COBOL. Indeed, prior to its addition, only the ADD, SUBTRACT, MULTIPLY, and DIVIDE verbs were available.

The ADD and SUBTRACT verbs offer the programmer flexibility in their use. For example, receiving operands can have their values altered in the operation using TO and FROM, or their values can be preserved using GIVING. Further, both verbs can accept multiple-sending operands and receiving operands.

The ADD instruction is even further distinguished by its ability—using the CORRESPONDING option—to perform group operations. Numeric variables that are identically named can be added using a single instruction rather than requiring a separate ADD instruction for each variable.

The MULTIPLY and DIVIDE instructions carry out their respective operations with fewer options than the ADD and SUBTRACT features. Here the only choices available involve whether the second field is to have its original value modified in the operation or not. If the original value is important, then the GIVING option will preserve that value.

The COMPUTE verb offers the programmer the ability to create arithmetic expressions containing more than one arithmetic operator. These operators, however, have been assigned priorities that dictate when and in what order they are executed by the computer. For this reason, programmers are encouraged to organize each of the subexpressions in their COMPUTE verb with enclosing parentheses. This guarantees that the subexpressions will be executed in the order desired by the programmer.

All of the numeric variables used thus far are said to be external in terms of their representation within computer storage. While external variables are less efficient than their internal counterparts, they are easier to define. Unfortunately, these external numeric variables make it difficult to accommodate negative values. On input each negative value's extreme right-hand digit must be either a letter of the alphabet or a blank, rather than a numeric digit. On output that same digit will appear as a blank or a letter of the alphabet. Problems of interpreting a report containing negative values is likely unless corrective measures are exercised.

On output, numeric fields that contain special editing characters can be defined. When negative values are placed into such fields, the extreme right-hand digit will actually print as a numeric digit, but the negative sign will be lost. To include the negative sign, a field suffix can be appended to the field to clearly show that the field contains a negative number.

Truncation can be avoided if the programmer recognizes the importance of defining numeric fields large enough to contain the greatest value a program is likely to generate. Loss of the extreme left-hand digits in a numeric field will not cause the offending program to abnormally terminate. Instead, a smaller numeric value will be printed.

Notes

1. Refer to an introductory textbook in data processing for a more complete discussion about the internal representation of data in computer systems.
2. In an EBCDIC byte (such as FO), the letter *F* is said to occupy the left nibble while the *O* occupies the right. A byte is composed of two nibbles.

Questions for Review

1. Which group of arithmetic statements—ADD, SUBTRACT, MULTIPLY, and DIVIDE **or** COMPUTE—is easier to use? Why?
2. Recall the ADD statement used to increment the page number in previous examples. Which statement is easier to use for this purpose?
3. In the ADD instruction, what is the difference between the TO and GIVING options (on the receiving operand)?
4. Why is the option of multiple operands attractive in the ADD and SUBTRACT instructions?
5. Why must users of the COMPUTE statement be wary of the use of multiple arithmetic operators?
6. Assume that an external numeric field without any special editing characters contains −1000. This field is to be printed. What value will appear on the printed report? Why? Could this confuse the reader? Why?
7. Is it a better strategy to oversize numeric fields or not? Why?
8. When the GIVING option was illustrated in this chapter, the receiving field named was always a numeric field that contained special editing characters. Since fields that contain special editing characters cannot be used as either the first or second operands of an arithmetic verb (such as ADD), what impact will accumulating grand totals of this receiving field have?

Problems

1. Examine the following arithmetic formula:

$$D = B - (E * C) + ((A / C) - E)$$

You are to write the COBOL instructions that would produce in D the value desired, using:

a. only ADD, SUBTRACT, MULTIPLY, and DIVIDE instructions
b. the COMPUTE instruction

2. What, if anything, is wrong with the following arithmetic statements?

a. `ADD 1 TO 2.`
b. `ADD 1 TO 2 GIVING C.`
c. `MULTIPLY A BY C GIVING D.`
d. `SUBTRACT A FROM Q GIVING B.`
e. `ADD 1 TO 2 GIVING 3.`
f. `DIVIDE C BY B.`

Convert as many of these instructions as possible to their COMPUTE verb counterparts.

3. Convert the following negative values to a more acceptable format for placement into an external numeric field.

−102, −2550, −8, −16

4. Examine the following partial record structures.

```
01  STRUCTURE-1.
    02 B            PICTURE IS 9(4).
    02 D            PICTURE IS 9V9.
    02 A            PICTURE IS 9(5)V99.
01  STRUCTURE-2.
    02 A            PICTURE IS 9(7)V99.
    02 B            PICTURE IS 9(10).
    02 C            PICTURE IS 9(2).
```

Construct the two alternative ADD instructions (i.e., CORRESPONDING and separate statements) to combine identically named variables and place the result in STRUCTURE-2.

If a variable named *C* were added to STRUCTURE-1, which of the two groups of ADD instructions would require modification? Why?

5. Examine the following data declarations and COMPUTE statement.

```
01  RECORDS-IN-ERROR      PICTURE IS 9(3).
01  RECORDS-READ          PICTURE IS 9(3).
01  PERECENT-INCORRECT    PICTURE IS 9(3).
    COMPUTE PERCENT-INCORRECT =
            RECORDS-IN-ERROR / PERCENT-INCORRECT * 100.
```

Why would this COMPUTE statement cause the percentage computed to be incorrect?

Programming Assignments

1. Revise the Sierra Realty Systems assignment from chapter 10 so that the report produced is like that shown in figure 11.25.

2. Thomas Information Systems offers both batch and on-line, interactive access to large computer users in San Antonio, Texas. The operating system software at the Thomas data center captures traffic data that detail both the type of computing resource as well as the quantity of the resource used. Each traffic record appears as follows:

Record Position	*Data Type*	*Variable*
01–08	AN	Computer user's identification
09–14	N	Date of record (YYMMDD)
15–22	N	Time of usage (HHMMSSTH)
23–24	N	Resource code
25–36	N	Quantity of resource used
37–44	N	Duration of use (HHMMSSTH)
45–52	AN	Activity identification

Different resources feature different charges. A list of each resource and its associated charges follows.

Resource Code	*Resource Type*	*Charge*
00	Processor Use	$100/hour
01	Primary Memory	$.10/1K/.01 sec
02	High Speed I/O	$.15/100
03	Low Speed I/O	$.30/100
04	Dedicated DASD	$10/minute
05	Dedicated Tape	$20/minute
06	Dedicated Unit Record Device	$5/minute

The records in the Traffic Accounting File are arranged by user, within user by date, and within date by resource used. A report program that produces a report like that shown in figure 11.26 is required.

Figure 11.25
Sierra Realty Systems revised sales report

NOTE: change in office or broker produces a new page

REPORT LAYOUT FORM

PROGRAMMER ______
PROBLEM ______
DATE ______
PAGE ____ OF ____

```
PIS-095                                   SIERRA REALTY SYSTEMS INC                               PAGE-1,002
DATE 02-15-80                            PROPERTY INVENTORY SYSTEM                                OFFICE 103
                                           LISTING/SALES REPORT                                   BROKER  18

   SALES    OFFICE                       DATE  OF              PRICE
  OFFICE    BROKER   SALES                 LIST                ASKED  PROPERTY
  NUMBER    NUMBER   ASSOC     CLASS       SALE                 SOLD  ADDRESS

     103        18      22     R       12-20-79         $104,999.00  130 W 7TH AVE, DENVER
                                       01-05-80           96,500.00
                                                           5,790.00
                                                                06%

                       105     C       12-20-79         $275,500.00  HENRY BLDG, 1400 GLACIER WAY, DENVER
                                       00-00-00                0.00
                                                          19,250.00
                                                                07%

                       111     L       12-22-79           $3,500.00  1 ACRE, LOT 84ZP-S-10, DENVER
                                       00-00-00                0.00
                                                             350.00
                                                                10%

                     COMMISSION INCOME - ACTUAL     $           5,790.00
                                         PROJECTED  $          19,600.00

                     VALUE PROPERTY - SOLD          $         104,999.00
                                      UNSOLD        $         279,000.00
```

Figure 11.26
Thomas Information Systems
traffic accounting report

PROGRAMMER ____________

PROBLEM ____________

DATE ____________

REPORT LAYOUT FORM

PAGE ____ OF ____

THOMAS INFORMATION SYSTEMS

TRAFFIC ACCOUNTING SYSTEM

USER RESOURCE ACCOUNTING

USER ID	DATE TIME	RESOURCE USED	UNITS USED	DURATION OF USE	COST OF RESOURCE
F699169	02-18-80	PROCESSOR USE	20	00:00:20	.56
	08:30:18	PRIMARY MEMORY	131,072	00:01:14	15.79
		HIGH SPEED I/O	114,090	00:00:00	17.11
		LOW SPEED I/O	12,180	00:00:00	3.65
		DEDICATED TAPE	247	00:04:07	5.00
		DEDICATED TAPE	259	00:04:19	5.00
		UNIT RECORD DEVICE	668	00:11:08	6.68
					$53.79
	02-18-80	PROCESSOR USE	5	00:00:05	.14
	10:11:00	PRIMARY MEMORY	65,536	00:00:55	5.87
		HIGH SPEED I/O	20	00:00:00	.00
		LOW SPEED I/O	500	00:00:00	.15
		UNIT RECORD DEVICE	10	00:00:10	.83
					$6.99
			**	TOTALS FOR F699169	$60.78 **
F699170	02-18-80	PROCESSOR USE	1	00:00:01	.03
	14:20:18	PRIMARY MEMORY	8,192	00:00:35	.47
					$.50
			**	TOTALS FOR F699170	$.50 **
			**	TOTAL TRAFFIC	$61.28 **

Sequential File Processing
Edit Program Structure

12

Chapter Outline

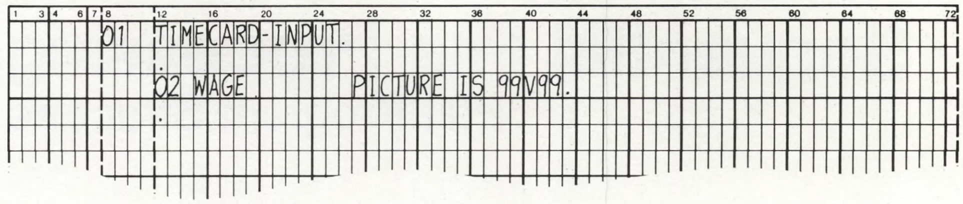


Figure 12.1
Record description for payroll application

Report programs are designed and constructed with the assumption that the input file of records contains error-free data. It is hoped, for instance, that each numeric field present in every input record contains numeric values. After all, arithmetic instructions can operate successfully only if the fields they are to manipulate contain the numeric digits 0 through 9. The *data validity* is not limited to whether or not numeric fields contain numeric digits, however. The problem is one of even greater magnitude.

Data Validity
The question in edit programs on whether or not a field contains the proper value.

In the payroll report program described earlier in this text, each input record represented an employee's time card. That time card contained, among other items, the employee's hourly wage. Wage was identified as an essential ingredient in the payroll computation. Wage is a numeric field, and if it is to be involved in an arithmetic operation, it must contain a legitimate numeric value. If the value is numeric, it does not necessarily mean that the numeric value present in the numeric field is a correct one.

Assume that wage is defined in a COBOL program as shown in figure 12.1. Note that the largest numeric value that WAGE can contain is $99.99 (editing characters added for clarity), and the smallest value possible is $00.00. This means that WAGE has a possibility of holding 10,000 different values ($00.00, $00.01, $00.02 . . . $99.99). For example, each employee is assigned an hourly wage. That employee's time card can contain one of the 10,000 possible numeric values in the field known as WAGE. However, only one of those 10,000 values is correct, even though all 10,000 are numeric. Consequently, it is not enough that a numeric field contain numeric digits, because the numeric value must also be the correct one.

Use of Edit Programs in System Design

Since computers themselves do not generate the transactions upon which they must operate, automated data processing systems are designed to include edit programs. The typical system design has a *source document* completed by a member of the organization. That document is then transcribed onto a machine-processable medium by a member of the organization's data entry staff. The first computer program to access the file of manually processed transactions—as just described—is the edit program.

Source Document
The document produced to record a transaction. It usually is not directly machine-readable and must first be converted to machine-readable form.

Rather than process each transaction immediately after it is placed on a machine-readable medium, these transactions are held in batches. A batch of transactions is submitted for computer processing at specific intervals of time. The amount of time that must elapse before a batch is processed depends on the nature of the application area from which the transactions come. For example, two applications with drastically different time periods are payroll and inventory.

Employees do not receive paychecks each day they report to work. Instead, paychecks are issued at the end of a week, every two weeks, or possibly at the end of each month. If paychecks are issued only once a month, the weekly time cards each employee generates need not be processed immediately. The time card from the first week of the month may not be processed until the third week of the month. After all, the hours from that pay period will be reflected in the check issued at the month's end.

Inventory applications are often more time critical. Sales transactions record items that have been removed from the inventory. These transactions can also be used to determine how an inventory is to be restocked. The longer the delay in replacing items sold, the greater the possibility of a customer desiring an item that is out of stock. In order to avoid this problem, the alternatives are (1) to keep greater levels of stock on hand than might be desirable; or (2) to process sales transactions at the end of each day, at which time replacement orders could also be sent to supplying vendors.

Since transactions are processed in batches and many of those transactions will contain errors, data entry personnel must carefully study the reports produced by edit programs. These reports, at a minimum, will show which transactions are in error, and the nature of each error. The transaction in error must be physically located and corrected. The corrected transactions can be processed as a separate batch or replaced in the batch from which they originated. In any event, the batch containing the corrected transactions must be processed by the edit program one more time. The correction process is a manual one and may introduce new errors into the transactions that were repaired.

Once a batch is free of errors, these transactions can be used to update payroll files, inventory files, and the like. These updated files can be used by report programs to generate reports to issue paychecks, send orders to vendors, and so on.

Posture of Edit Programs

All edit programs are designed to find input records that contain incorrect data values. In fact, that is the only function of an edit program. Not only must numeric fields contain numeric digits, but all fields must contain an appropriate value for the record being processed.

Many data processing systems operate on transactions that record fundamental business activities. These activities include payroll, and where appropriate, inventory processes. Despite the increasing interest in office automation, many of these transactions are recorded manually onto special printed forms called source documents. Source documents contain all the data describing the transactions they represent. Source documents are designed for human, not machine, use.

When semiautomated and automated processing techniques are introduced into business activities that have previously been done manually, it is necessary to convert the data from source documents into machine-readable form. This transcription, or conversion, process introduces a new level of processing in which errors can be introduced. The employee who records a transaction onto a souce document might do so with one or more errors. The transcriber who converts the transaction into a form usable by automated processing techniques might also do so with one or more errors. Consequently, errors can be introduced at two points in the processing of the transaction. The edit program examines each record, field-by-field, to insure that the record accurately represents the transaction.

Why is the quality of transactional data processed by automated systems so important? The answer lies in the fact that bad data propagates.

If payroll systems did not possess edit programs, time cards containing incorrect numeric values would be processed and the error or errors would not be noticed. For example, if a time card lists an hourly wage far above the correct rate, that person's pay for the payroll period will be much greater than it should be. If the employee who receives a larger paycheck than expected does not return the check for one with the correct amount, the error will be compounded in all accounting reports that represent operating expenses, like payroll. Even the organization's profit and loss statements will be based on those incorrect paychecks.

General Structure of Edit Programs

The task of each edit program is very clear—to examine each field in every record. Records containing one or more fields of invalid data should be printed along with an error message that clearly explains the nature of each error that is found. In fact, summary statistics that describe the total number of errors—and other items of interest—can help determine the frequency and, possibly, the source of errors.

To start, you must realize that the file of records—such as time cards for a payroll application or inventory transactions for an inventory application—must have its records read and processed one at a time. After a record has been read, each field in that record should be examined. The general structure of our edit program, in pseudo-coding, must be that shown in figure 12.2. As you can see, this program's logic checks the contents of only two fields. This is because the logic is designed to edit records containing only two fields. What if the records contain many more fields to be checked? Then statements 6, 6a, and 6b would be repeated for each field that the records possess. A record containing twenty-five fields would have a pseudo-coded structure with twenty-five sets of statements like statement 6 and statement 7.

Note that the main control portion of our pseudo-coded logic (statements 1 through 3) is similar to that for report programs. First, a record is read to start the process that is the object of the iterative statement. Second, an iterative statement causes a group of instructions to be repeatedly executed until the input file is emptied. Third, when the input file's terminator record is encountered, the program concludes its execution.

Report Titles, Trailers, and Pagination

The first version of our edit program logic does not reflect the earlier discussions of such things as report titles. Now that the fundamental structure of the edit program is clearly established, it is time to add these refinements to yield a more useful programming model. Note the additions to the earlier version of the edit program shown in figure 12.3. Now the error report produced by the edit program will contain page titles and trailers on each page of the report.[1] No more than one page of lines will ever be printed before a page overflow condition is sensed and trailers and titles are printed.[2]

A Basic Statistic

One statistic our edit program should compute is a count of the total number of records containing one or more errors. If all the fields in an input record contain acceptable data, then this count should not be altered. However, if one, two, or more of a record's fields are in error, then this count should be increased by one.

If the number of records containing one or more errors is known, then another statistic can be derived. That statistic is the percentage of all records in the input file that are in error (i.e., contain one or more fields in error). This task is a little more complex than it might first appear.

How does the edit program know when a record being processed is in error? The answer is when it finds a field in that record containing incorrect data. If even one field is incorrect, the entire record is in error. Remember, however, each field is examined one at a time. You might be tempted to construct a set of pseudo-coded statements as shown in figure 12.4.

Assume that a record contains only two fields and both fields contain correct data. In this case, no error will be detected, the record will not be considered in error, and the bad record count will remain unchanged. On the other hand, assume the next record in the file has one field with correct data while the remaining field is incorrect. This means that one of the inquiry statements will be false (i.e., the field is correct), the second inquiry statement will be true (i.e., the field contains an error), statement 10c will not be executed, statement 11c will be executed, and the bad record count will be incremented by one.

Figure 12.2
Report model modified for editing

```
Statement    Pseudo-coded statement
number

1                Read the very first record to be
                     checked field by field
2                Perform the field-by-field edit
2a                   until
2b                       the file is empty
3                Stop the program's execution

4        Field-by-Field Edit
5                Print the record just read
6                If field 1 is invalid
6a                   then
6b                       Print appropriate error message
7                If field 2 is invalid
7a                   then
7b                       Print appropriate error message
8                Read next record to be checked
```

Figure 12.3
Basic edit model

```
Statement    Pseudo-coded statement
number

1                Print report titles
2                Read the very first record to be
                     checked field by field
3                Perform the field-by-field edit
3a                   until
3b                       the file is empty
4                Print report trailers
5                Stop the program's execution

6        Field-by-Field Edit
7                If page overflow
7a                   then
7b                       Print report trailers
7c                       Print report titles
7d                       Reset line counter
8                Increment line count by 1
9                Print the record just read
10               If field 1 is invalid
10a                  then
10b                      Print appropriate error message
11               If field 2 is invalid
11a                  then
11b                      Print appropriate error message
12               Read next record to be checked
```

Figure 12.4
Counting fields in error

```
Statement    Pseudo-coded statement
number

6        Field-by-Field Edit
                 .
                 .
                 .
10               If field 1 is invalid
10a                  then
10b                      Print appropriate error message
10c                      Increment bad records found by 1
11               If field 2 is invalid
11a                  then
11b                      Print appropriate error message
11c                      Increment bad records found by 1
12               Read next record to be checked
```

Figure 12.5
Counting bad fields rather than records

Statement number	Pseudo-coded statement
6	Field-by-Field Edit
	.
	.
	.
10	If field 1 is invalid
10a	then
10b	Print appropriate error message
10c	Increment bad fields found by 1
11	If field 2 is invalid
11a	then
11b	Print appropriate error message
11c	Increment bad fields found by 1
12	Read next record to be checked

Finally, assume that the third record in our file contains incorrect data in both fields. In this case both inquiry statements will be true (i.e., the fields are bad), statements 10c and 11c will be executed, and one record will be counted in error twice. This is not what we wanted to accomplish. Records with more than one bad field should be counted only once, yet the inquiry statements are structured to count a record as being in error once for each bad field it contains. Thus, we are not counting records in error. Instead, we are counting the number of errors found. The number of errors found may be a valuable statistic; however, it is not the one we are currently attempting to derive. The revised pseudo-code in figure 12.5 shows that bad fields are being counted, not bad records.

We will now determine a method that will allow the count of the number of records in error. However, assuming that the number of errors found is a worthwhile statistic to have, our pseudo-coding will be altered to more correctly state that this count is the statistic being derived.

Bi-stable Devices

Our program's logic must be altered to remember an error in a field each time one is found. If our program can remember that at least one error was found, then a question can be asked before reading the next record. That question is: "Was an error found while processing the current record?" If the answer is yes, our program can be directed to increment a bad record count by one. If not, the bad record count remains unchanged. The method used to remember whether an error was found while processing the current record is known as a *bi-stable device.*

Bi-stable Device
A COBOL variable that can take on one of two values. One value indicates that an anticipated condition has occurred. The other value indicates that the condition has not yet occurred.

Bi-stable implies that the device has two states. Either it remembers, for instance, that an error has occurred or it has no such knowledge, because an error has not occurred. In other words, the device responds either yes or no to the inquiry: "Did we encounter one or more bad fields in the record currently being processed?" Examine figure 12.6, the most recent version of our pseudo-coded program logic, to see how such a device is actually used. When examining our most recent version of the edit program's logic, it should be clear that our original structure of processing still remains as follows: (1) a record is read; (2) that record's fields are examined one at a time; (3) every time a bad field is found, an instruction to increment the number of errors encountered is executed; and (4) when the record has been completely examined, a new record is read and the process starts anew.

Statement number	Pseudo-coded statement
6	Field-by-Field Edit
7	If page overflow
7a	then
7b	Print report trailers
7c	Print report titles
7d	Reset line counter
8	Increment line count by 1
9	Print the record just read
10	Reset bi-stable device to "no"
11	If field 1 is invalid
11a	then
11b	Print appropriate error message
11c	Increment bad field count by 1
11d	Set bi-stable device to "yes"
12	If field 2 is invalid
12a	then
12b	Print appropriate error message
12c	Increment bad field count by 1
12d	Set bi-stable device to "yes"
13	If bi-stable remembers "yes"
13a	then
13b	Increment bad record count by 1
14	Read next record to be checked

Figure 12.6
Counting bad records

Unlike earlier versions, however, if a field is found to contain unacceptable data, the bi-stable device is set to show an error has been found. This can occur in statements 11d and 12d. When all the fields have been checked, an inquiry is directed to the bi-stable device. That inquiry is: "Was an error found during the processing of this particular record?" If the answer is yes, the bad record count is incremented by one. The answer can only be yes if either statement 11d or 12d is executed. Either statement can only be executed if an incorrect field was found.

Also note that statement 10 resets the bi-stable device to no with each new record read. Each record starts with no memory that an error has taken place. The only way the bi-stable device can have its memory altered from the no state to the yes state is if an error is found in the processing of this new record.

Summary

Business transactions are traditionally handled manually by employees in an organization. These transactions can be recorded with or without error. The use of automated data processing systems requires that the data present on the source document be placed in machine-readable form. This transcription activity can also introduce error.

The possibility of error causes the designers of automated data processing systems to examine each transaction prior to its intended use. The examination mechanism is an edit program. Edit programs check each field in every record to be processed.

The pseudo-coded version of the logic of an edit program is iterative in nature—records are read and checked one at a time until no more records remain to be processed. This basic, logical structure can easily be improved to print the now familiar page titles and trailers, where appropriate. In addition, basic statistics describing the characteristics of all records processed can be obtained.

The total number of errors found is easily implemented. Each time a field is checked and found to be in error, this numeric count can be incremented by one. For example, a record with three of its ten fields in error would cause the numeric count to be incremented by three (number of errors found). This same record, however, would cause the count for the number of records with any errors found to be incremented by only one.

Figure 12.7
Final model of edit program

```
Statement      Psuedo-coded statement
 number

  1            Print report titles
  2            Read very first record to be checked
                    field by field
  3            Perform field-by-field check
  3a                until
  3b                     there are no more records
                         to check
  4            Compute statistics
  5            Print statistics
  6            Print report trailers
  7            Stop program execution

  8       Field-by-Field Check
  9            If page overflow
  9a                then
  9b                     Print report trailers
  9c                     Print report titles
  9d                     Reset line counter
 10            Increment line count by 1
 11            Reset bi-stable device to "no"
 12            Print record just read
 13            If field 1 is invalid
 13a                then
 13b                     Print appropriate error message
 13c                     Increment errors found by 1
 13d                     Set bi-stable device to "yes"
 14            If field 2 is invalid
 14a                then
 14b                     Print appropriate error message
 14c                     Increment errors found by 1
 14d                     Set bi-stable device to "yes"
 15            If bi-stable device is "yes"
 15a                then
 15b                     Increment bad records found by 1
 16            Read next record to be checked
```

Keeping track of this last statistic, the number of records with one or more errors, is somewhat more difficult. Our program logic must be equipped with a bi-stable memory device that can remember whether the record currently being processed has any fields in error. If the answer to such an inquiry is yes, the count is incremented. If not, then all fields in the record were correct and the count of records with one or more errors remains the same.

Finally, the complete pseudo-coded version of the edit program logic is shown in figure 12.7. Recall that the number of fields to be checked will vary from application to application. Here the records possess only two fields to be checked.

Notes

1. You have undoubtedly noticed that the pages in your reports do not coincide with the perforations in the actual paper stock. This is one of the shortcomings of the DISPLAY verb. In a later chapter a new output instruction—WRITE—will be introduced. This new verb will permit reports to be properly positioned on the paper stock so that the perforations appear above the titles and below the trailers.
2. A question might arise, however, regarding the choice of page size that should be used in statement 8 in the program logic. Clearly, every record read will be printed. Yet, how many error messages will be printed for each record? Some will require none, others only one. Still others will produce an error message for each field they contain.

Questions for Review

1. Why are edit programs necessary? What could happen if edit programs did not exist?
2. Why is it insufficient for a numeric field to contain numeric digits to pass an edit?
3. All discussions of field edits have centered on numeric fields. Do nonnumeric fields have edit criteria? Give an example.
4. Why is figure 12.4 incorrect?

Problems

1. What, if anything, is wrong with the following section of pseudo-coded statements?

```
If field-3 is invalid
    then
        Print appropriate error message
        Increment errors found by 1
        Increment records in error by 1
```

2. What, if anything, is wrong with the following pseudo-coded version of an edit program's logic?

```
        Read very first record to be checked
            field by field
        Perform field-by-field check
            until
                there are no more records to check
        Print report titles
Field-by-Field Check
        If page overflow
            then
                Print report trailers
                Print report titles
        Print record just read
        If field 1
            then
                Print appropriate error message
                Increment errors found by 1
                Set bi-stable device to "yes"
        If field 2
            then
                Print appropriate error message
                Increment errors found by 1
                Set bi-stable device to "yes"
        If bi-stable device is "yes"
            then
                Increment records with errors by 1
        Read next record to be checked
```

13 Sequential File Processing
Implementing Edit Programs

Chapter Outline

The programming methodology in this text consists of two parts. The first part involves identifying the logical structure of a major program type, like the report program, by using pseudo-code. Following the complete definition of a program's operation, we can then convert each statement in the pseudo-coded program to its COBOL equivalent.

Translating a pseudo-coded program model to COBOL requires attention to detail. The DATA DIVISION must include record descriptions for titles, trailers, and other inputs and outputs; the PROCEDURE DIVISION must contain the appropriate sections and the necessary COBOL verbs within those sections.

At this point, you know enough COBOL to construct a working COBOL edit program. However, edit programs often make interesting demands on the programmer. A complex *field validity test* can lead to equally complex programs. Unfortunately, complex programs are a problem—they are difficult to construct, and they are even more difficult to maintain once they become operational.

Field Validity Test
To examine a field that is usually part of a larger record structure to determine whether the data contained within is correct or incorrect.

The function of this chapter is twofold. First, we will review the COBOL necessary to implement the pseudo-coded edit program model you learned earlier. Second, new COBOL statements will be introduced to make complex, and often confusing, edit programs easier to construct and maintain.

Edit Program Model

The pseudo-coded logic that is common to all edit programs was developed in chapter 12. This program reads one transaction at a time and examines each of the transaction's fields for possible error. Every field in error causes an appropriate error message to be printed, as well as appropriate error statistics to be displayed at the program's conclusion. This process of reading and examination continues until the entire file of transactions to be edited has been read and examined.

The main-control portion of our edit program logic is very similar to the main-control portion of the report program logic. In both program types, the process continues until the input file is exhausted. Since you have already learned how to implement the entire report program model into COBOL, you should have no trouble converting the main-control portion of the pseudo-coded edit program in COBOL as well.

The field-by-field examination is unique to the edit program model. This segment of the pseudo-code is reproduced in figure 13.1 from chapter 12. Page overflow that was shown at the beginning of figure 13.1 has been encountered many times in our report programming discussions earlier in the text. Here, as with main-control, this series of pseudo-coded statements can be translated into COBOL quite easily. The concept of bi-stable entities is a new topic.

Bi-stable Devices

A bi-stable device is needed in our edit program to detect whether or not the current transaction being processed features one or more errors. This device is needed because our edit program monitors both the total number of errors found and the total number of records featuring one or more fields in error.

In the abstract, a bi-stable device is easy to understand. Light switches are either on or off. Similarly, the transaction being examined either possesses an error or it does not. Remember, a bi-stable device contains only one of two possible values at any point in our program's execution. Either an error has been found (YES) or no field in our current input record has been found to be in error (NO).

Figure 13.1
Field checking in edit program model

Statement number	Pseudo-coded statement

```
 8     Field-by-Field Check
 9             If page overflow
 9a                then
 9b                    Print report trailers
 9c                    Print report titles
 9d                    Reset line counter
10             Increment line count by 1
11             Print transaction just read
12             Reset bi-stable device to "no"
13             If field 1 is invalid
13a                then
13b                    Print appropriate error message
13c                    Increment errors found by 1
13d                    Set bi-stable device to "yes"
14             If field 2 is invalid
14a                then
14b                    Print appropriate error message
14c                    Increment errors found by 1
14d                    Set bi-stable device to "yes"
15             If bi-stable device is "yes"
15a                then
15b                    Increment transactions with
                          errors by 1
16             Read next transaction to be checked
```

Figure 13.2
Defining and manipulating bi-stable device

Bi-stable variable definition

```
       01  TRANSACTION-IN-ERROR    PICTURE IS X(03) VALUE "NO".
```

Setting bi-stable to reflect error (13d, 14d)

```
           MOVE "YES" TO TRANSACTION-IN-ERROR.
```

Resetting bi-stable to show no error (12)

```
           MOVE "NO" TO TRANSACTION-IN-ERROR.
```

A possible COBOL bi-stable variable in the WORKING-STORAGE SECTION can be found in figure 13.2. As defined here, our bi-stable device has been given an initial value of NO. No field in the record has been found to be in error. To change the state, or meaning, of this variable, the programmer need only insert the appropriate instruction into the PROCEDURE DIVISION, as noted in figure 13.2. This particular MOVE instruction is somewhat different from earlier versions. The sending field in the instruction is not a variable from the program's WORKING-STORAGE SECTION. Instead, the sending field is the nonnumeric literal YES. (The numbers in parenthesis following the instructions in figures 13.2, 13.3, and 13.4 identify the pseudo-coded statements shown in figure 13.1.)

Consequently, the variable in figure 13.2 contains NO before a transaction is read. The MOVE instruction replaces NO with YES when an error is detected. If, after all of the fields in a transaction have been examined, TRANSACTION-IN-ERROR still contains NO, the transaction possesses no incorrect fields. However, if the value is YES, one or more of the transaction's fields are in error. The value YES does not tell us which field or fields are bad, but it does tell us that at least one error was detected.

Bi-stable variable definition

```
       01  TRANSACTION-IN-ERROR     PICTURE IS X(03) VALUE "NO".
```

Inquiry statement (15) in pseudo-code

```
           IF TRANSACTION-IN-ERROR = "YES"
               THEN
                   ADD 1 TO TRANSACTIONS-FOUND-IN-ERROR.
```

Figure 13.3
Incrementing count of records in error

Bi-stable variable definition

```
       01  TRANSACTION-IN-ERROR     PICTURE IS 9(01) VALUE 0.
```

Setting bi-stable variable (13d, 14d)

```
           MOVE 1 TO TRANSACTION-IN-ERROR.
```

Resetting bi-stable value (12)

```
           MOVE 0 TO TRANSACTION-IN-ERROR.
```

Inquiring as to value (15)

```
           IF TRANSACTION-IN-ERROR = 1
               THEN
                   ADD 1 TO TRANSACTIONS-FOUND-IN-ERROR.
```

Figure 13.4
Numeric bi-stable variable

Sending fields in a MOVE instruction can be literals. If the receiving field is a numeric field (PICTURE IS 9), the literal must be a numeric literal without delimiting quotes and with numeric digits. On the other hand, if the receiving field is a nonnumeric field, as is the case with TRANSACTION-IN-ERROR, the literal must be a nonnumeric literal delimited by quotes. Thus, statements 13d and 14d from the earlier pseudo-coded version of the edit program logic would use the MOVE instruction described in figure 13.2.

Statement 12 in figure 13.1 plays a special role in the edit program. Note that as of statement 11 we have not yet begun to examine the current transaction; in fact, the examination process actually starts in statement 13. TRANSACTION-IN-ERROR, our bi-stable device, should contain NO before the first field in each transaction that is examined. However, the last transaction inspected might have been in error, causing TRANSACTION-IN-ERROR to contain YES. Consequently, we must automatically reset TRANSACTION-IN-ERROR, in statement 12, to show that no error has yet been found. This is also carried out by a MOVE instruction, as illustrated in figure 13.2.

To inquire whether or not the bi-stable device is YES requires the use of the IF statement in COBOL. This statement has been previously used for monitoring page overflow in our report programs. As a result, statement 15 from the pseudo-coding appears in COBOL as shown in figure 13.3.

Bi-stable devices can be implemented in COBOL as numeric variables as well as nonnumeric variables. The only thing to remember is that the programmer must define a variable in the WORKING-STORAGE SECTION of the COBOL program that is capable of holding one of two values. If not YES or NO, what of the values one or zero? Figure 13.4 illustrates a numeric bi-stable variable. The only problem with using a numeric variable for a bi-stable device is that we can easily forget whether a value of zero represents YES or NO. A nonnumeric variable can be defined to hold values that clearly state whether an error has been found or not. If the value is YES, an error was found. If the value is NO, an error was not found. What does zero represent? Remember, program clarity is a primary concern of the programmer. Nonnumeric bi-stable variables help improve clarity, and the meanings of YES and NO are not likely to cause confusion. However, if you must implement a bi-stable device using values that do not clearly indicate the status of the transaction being processed, COBOL provides an aid.

Condition Names

Assume that you have chosen to define a bi-stable device as a numeric variable in your COBOL program. Statement 15 from the pseudo-coded version of the edit program logic might appear as shown in figure 13.4. It may not be clear to the reader of your program whether a one means a field has been found in error. To avoid this problem, COBOL permits us to define variables and also to symbolically name the values those variables hold. This symbolic name is termed a *condition name*. For instance, note the somewhat different appearance of TRANSACTION-IN-ERROR in figure 13.5.

Condition Name
A symbolic name for a particular value that a variable can contain. When the variable contains the value specified, then the condition is said to exist.

Items defined as *88-level* are not variables but are symbolically named conditions. A condition is a particular value in a specific variable. The conditions defined are assigned names using the same rules that are used to construct variable names. Consequently, the condition TRANSACTION-IS-GOOD can occur only if the variable TRANSACTION-IN-ERROR contains zero. Should TRANSACTION-IN-ERROR ever possess a value of one, the condition TRANSACTION-IS-BAD exists. TRANSACTION-IN-ERROR holds either a zero or a one. It cannot contain both simultaneously. Correspondingly, either TRANSACTION-IS-GOOD or TRANSACTION-IS-BAD exists. It cannot be both.

Statement Format

```
B
↓
88  conditionname     VALUE literal.
```

The 88-level must begin somewhere in the B-Margin, without a PICTURE IS clause. (Only the VALUE clause is used.) Once again, conditions are not variables in a traditional sense. They are not allocated memory space in the record descriptions where they are used. Instead, they are symbolic definitions of values a variable may hold. The variable in this case is TRANSACTION-IN-ERROR.

Condition names permit us to construct more meaningful inquiry statements. Statement 15 from the pseudo-coded version of the edit program logic can now be written as shown in figure 13.6. This IF statement is quite clear about what is being tested. This IF can only be true if, during the course of processing the current transaction, one or more fields were found in error. If so, then TRANSACTION-IS-BAD.

Statement Format

```
B
↓
IF conditionname
   THEN
       statement(s).
```

```
       01  TRANSACTION-IN-ERROR       PICTURE IS 9(01) VALUE 0.
           88 TRANSACTION-IS-GOOD     VALUE 0.
           88 TRANSACTION-IS-BAD      VALUE 1.
```

Figure 13.5
Variable definitions with condition values

```
           IF TRANSACTION-IS-BAD
                 THEN
                       ADD 1 TO TRANSACTIONS-FOUND-IN-ERROR.
```

Figure 13.6
Testing for record error using condition values

Condition Tests
The use of a condition name (identified in an 88-level) in an IF statement to determine whether the condition exists.

Condition tests are somewhat different from the earlier forms of the IF statement. Notice that a condition test does not make use of any relational operators such as equal, not equal, greater than, etc. In fact, if you try to use a relational operator with a condition name (i.e., an 88-level name), the COBOL compiler issues an error message when it attempts to translate your program.

A condition name is a symbolic definition of a literal value. In the earlier definition, TRANSACTION-IS-BAD was equated to the literal value 1 in the variable TRANSACTION-IN-ERROR. Hence, the IF statements in figures 13.5 and 13.6 are identical—they are testing for the same thing. They are equivalent because the numeric digit 1 in the field named TRANSACTION-IN-ERROR is the same as TRANSACTION-IS-BAD.

Condition Variables

Greater program clarity can be achieved at other locations within our edit program through the use of symbolically named conditions. Detecting page overflow is an excellent example.

Statement Format

```
B
↓
88  conditionname      VALUE literal literal [literal...].
```

Until now, page overflow has been anticipated through a series of COBOL statements like the ones shown in figure 13.7 (i.e., a numeric variable, through its series of values, traces the positioning of the paper stock in the printer). As figure 13.7 shows, if the value of CURRENT-LINE exceeds 55, any further printing on the present page might either cause the report detail to:

1. appear in that part of the page reserved for trailer lines, or in the extreme,
2. overflow onto the next page where the title lines are to be placed.

Figure 13.7
Alternatives in condition names and variables

Variable definition and inquiry

```
       01  CURRENT-LINE        PICTURE IS 9(02) VALUE 0.

           IF CURRENT-LINE > 55
               THEN
                   PERFORM PRINT-TRAILERS
                   PERFORM PRINT-TITLES.
```

Alternative one

```
       01  CURRENT-LINE        PICTURE IS 9(02) VALUE 0.
           88 PAGE-OVERFLOW    VALUE 56.

           IF PAGE-OVERFLOW
               THEN
                   PERFORM PRINT-TRAILERS
                   PERFORM PRINT-TITLES.
```

Alternative two

```
       01  CURRENT-LINE        PICTURE IS 9(02) VALUE 0.
           88 PAGE-OVERFLOW    VALUE 56 THRU 99.

           IF PAGE-OVERFLOW
               THEN
                   PERFORM PRINT-TRAILERS
                   PERFORM PRINT-TITLES.
```

Examine alternative one shown in figure 13.7. Here a symbolic condition known as PAGE-OVERFLOW has been established. When the value in CURRENT-LINE is 56, the condition PAGE-OVERFLOW exists. The ability to comprehend the function of the IF statement has been greatly improved by the use of the condition name. Unfortunately, CURRENT-LINE might contain the value of 57 when the IF statement is executed. Clearly, a value of 57, which is greater than 55, indicates a page overflow. However, PAGE-OVERFLOW only exists while CURRENT-LINE contains 56. As a result, overflow is not detected and undesirable results occur.

Alternative two, on the other hand, allows the condition PAGE-OVERFLOW to be associated with any numeric value in the range of 56 to 99. Surely 99 is somewhat excessive, but it is better to be safe. Now whenever CURRENT-LINE takes on a value in the range identified, the *condition variable* exists, page overflow is detected, and the report detail is confined to the detail segment of the printed page.

Condition Variable
A symbolic name that represents more than one possible value a variable can contain. Condition names can be associated with only one value.

Statement Format

```
B
↓
88   conditionname     VALUE literal THRU literal.
```

Field Validity Checks

Guidance for converting all but two pseudo-coded statements in figure 13.1 has been offered. You have learned enough COBOL to monitor page overflow, read and print transactions, increment numeric variables, and monitor or manipulate bi-stable devices. Do you know enough COBOL, however, to check the validity of each field in a transaction? For some fields, yes; for other fields, no.

Some applications present transactions containing a name field. In an inventory application this could be the name of the item being stocked; in a payroll application it is likely to be the name of the employee. Both cases are a dilemma of sorts. What constitutes a validity check for a name field in an edit program?

```
           IF EMPLOYEE-NAME = SPACES
               THEN
                   DISPLAY ERROR-MESSAGE-2
                   MOVE "YES" TO RECORD-IN-ERROR
                   ADD 1 TO FIELDS-FOUND-IN-ERROR.
```

Figure 13.8
Determining the absence or presence of data

Alternative one

```
           02 HOURS-WORKED         PICTURE IS X(02).

               IF HOURS-WORKED > "00" AND
                  HOURS-WORKED < "99"
                   THEN
```

hours-worked is numeric and the field contains no error.

Figure 13.9
Multidigit numeric field tests

Alternative two

```
           02 HOURS-WORKED.
               03 POSITION-1       PICTURE IS X(01).
               03 POSITION-2       PICTURE IS X(01).

           IF POSITION-1 OF HOURS-WORKED > "0" AND
              POSITION-1 OF HOURS-WORKED < "9" AND
              POSITION-2 OF HOURS-WORKED > "0" AND
              POSITION-2 OF HOURS-WORKED < "9"
               THEN
```

hours-worked is numeric and the field contains no error.

Product and employee names come in an overwhelming variety. You cannot construct a program to verify proper spelling of a name, for instance, when the name is not known beforehand. What can be done? For the moment we can only determine whether or not the name field in a transaction is blank. If the name field is blank, then an error has been encountered; if it is not blank, no matter what nonblank characters it holds, no error exists.

Figure 13.8 illustrates the COBOL coding to examine a name field for a payroll application. Unfortunately, few validity checks can be accomplished as simply as this.

Examining Numeric Fields

Single-digit numeric fields are quite easy to examine. We can construct what is known as a range check. That is, we devise an IF statement that determines whether the digit is greater than or equal to 0 and less than or equal to 9. If the digit is in the range of 0 to 9, the field must contain a numeric digit.

Until the field passes this range check, we have no idea whether the contents of the field are numeric. After all, someone could have miskeyed a letter of the alphabet instead. Edit programs are required because transactions may contain bad data, and numeric fields may not contain strictly numeric data. Consequently, to be absolutely careful, in edit programs we define supposedly numeric fields as alphanumeric fields instead. The reason is simple. If the field is defined as numeric and contains one or more nonnumeric characters, an abnormal program termination may occur.

Examine the COBOL program excerpt in figure 13.9. This excerpt shows a supposedly numeric field named HOURS-WORKED. This is a multiple-digit field, so it stands to reason that each of the field's digits must be examined to verify that they are numeric. Unfortunately, alternative one defined HOURS-WORKED as a two-digit field. Access to each individual digit is not possible.

Alternative two presents a slightly different definition for HOURS-WORKED. Now access to each individual digit is possible. The IF statement in alternative two shows both digits being examined; however if alternative two is used, examining a larger numeric field, like Social Security number, is going to be very cumbersome. Each of the Social Security number's nine digits must be subjected to a numeric range test.

Multidigit numeric fields cannot be declared to contain numeric values by virtue of a single range check. Note the definition and IF statement in alternative one of figure 13.9. It appears that if HOURS-WORKED contains a value in the range specified, then the value is numeric. Unfortunately, this is not the case. The collating sequence used by the computer places a nonnumeric digit in the range 00 to 99. How can this be? How can a nonnumeric value be included in the range 00 to 99?

The sequence of digits in the range 00 to 99 starts with 00, 01, 02, . . . 09. What follows 09? The answer to this question requires some knowledge of the computer's collating sequence.

EBCDIC
The Extended Binary Coded Decimal Interchange Code, which is the collating sequence of characters used to represent data in a computer.

Certain, but not all, computers use the EBCDIC collating sequence. In the EBCDIC sequence, the representation of 09 is F0F9. What follows F0F9 is F0FA. F0FA is not a two-digit numeric value, yet F0FA is the next two-digit value to follow F0F9 (09) in the EBCDIC collating sequence. In fact, F0FA is followed by the following nonnumeric two-digit values: F0FB, F0FC, F0FD, F0FE, and F0FF. The value F0FF is followed, according to the collating sequence, by the numeric value F1F0. As a result, alternative one in figure 13.9 does not exclude nonnumeric characters. Still, a more attractive alternative than that shown in alternative two must be available.

Class Tests

Class Tests
Built-in COBOL functions that examine variables to see whether they are numeric (the numeric class test) or are alphabetic (the alphabetic class test).

COBOL features a special function that can examine each of the digits in a multiple-digit field to determine whether the value is numeric or not. This function is known as a *class test.* Figure 13.10 illustrates the use of one type of class test. By following a variable name with the word NUMERIC, COBOL will automatically check each of the digits in the field specified. If the value in this field is numeric, the IF statement is true; if it is not numeric, the IF statement is false.

COBOL also features another class test. This test is known as ALPHABETIC. A field is said to be ALPHABETIC if it contains any combination of the letters of the alphabet and/or spaces.

Statement Format

```
B
↓                              {ALPHABETIC}
IF variablename IS [NOT]       {NUMERIC   }
   THEN
        statement(s).
```

You might have noticed in the NUMERIC class test, however, that we do not want to know if the field contains good data. We want to know only if the field is bad.

```
           IF HOURS-WORKED NUMERIC
               THEN
                   hours-worked is numeric.
```

Figure 13.10
Use of class test to check multidigit numeric fields

True and False Outcomes

Until now, every IF statement written in COBOL has been a shortened version of the complete instruction. The general form of the IF instruction has been thought to be similar to the one shown in figure 13.11. If the condition being evaluated is true, control proceeds to the one or more statements associated with the THEN portion of the instruction; if the condition is false, the one or more statements in the THEN portion are skipped.

A more complete form of the IF statement is found in figure 13.12. The ELSE portion of the IF statement in COBOL has been totally ignored until this point.

Under what circumstances is the ELSE statement used? Examine the COBOL program segment from an edit program shown in figure 13.13. If HOURS-WORKED is found to be numeric, the IF statement produces a true outcome. True outcomes cause control to proceed to the THEN portion of the IF statement. The THEN portion, however, contains only one COBOL instruction—and a new instruction at that. The instruction NEXT SENTENCE causes control to leave the IF statement entirely. Remember, the IF statement is terminated by the first period found immediately after the word IF in the instruction. Consequently, control automatically passes to the instruction after the IF statement. If the outcome of the condition is false, control progresses to the ELSE portion of the IF statement, skipping the NEXT SENTENCE instruction completely. In this instance, we want control to cause the error message to print and indicate that HOURS-WORKED contains unacceptable data.

```
IF condition
   THEN
       one or more COBOL statements
       only the last of which is
       terminated by a period.
```

Figure 13.11
Short version of the IF statement

```
                       IF condition
                          THEN
executed if                   one or more COBOL
condition                     statements
is true                       with no periods whatsoever
                          ELSE
executed if                   one or more COBOL
condition                     statements
is false                      with the last statement
                              terminated by a period.
```

Figure 13.12
Complete version of IF statement

```
           IF HOURS-WORKED NUMERIC
              THEN
                  NEXT SENTENCE
              ELSE
                  MOVE "YES" TO TRANSACTION-IN-ERROR
                  ADD 1 TO FIELDS-FOUND-IN-ERROR
                  DISPLAY ERROR-MESSAGE-2.
```

Figure 13.13
Example of fully specified IF statement

Statement Format

```
B
↓
IF condition
   THEN
         NEXT SENTENCE
   ELSE
        statement(s).
```

Statement Format

```
B
↓
IF condition
   THEN
        statement(s)
   ELSE
        statement(s).
```

Examine the COBOL program segment shown in figure 13.14. Will the first version of the field check for HOURS-WORKED perform properly? The answer is yes. Printing an error message, setting the bi-stable device, and incrementing the number of errors found will only occur if HOURS-WORKED contains a nonnumeric digit. What about the second version shown? Here again, the error message is only activated if the field is nonnumeric. If the field is numeric, the condition is false, and control passes immediately to the ELSE portion of the IF statement. The ELSE portion, however, only causes control to exit from the IF statement to the next statement in the program. Actually, the last three examples of checking the field HOURS-WORKED produce identical results. The first and third examples, however, are somewhat awkward and may, in fact, diminish program clarity. The second example is more easily understood, and, consequently, more often used.

Figure 13.14
Alternatives in class tests

```
           IF HOURS-WORKED NOT NUMERIC
               THEN
                   DISPLAY ERROR-MESSAGE-3
                   MOVE "YES" TO TRANSACTION-IN-ERROR
                   ADD 1 TO FIELDS-FOUND-IN-ERROR.
```

or

```
           IF HOURS-WORKED NOT NUMERIC
               THEN
                   DISPLAY ERROR-MESSAGE-3
                   MOVE "YES" TO TRANSACTION-IN-ERROR
                   ADD 1 TO FIELDS-FOUND-IN-ERROR
               ELSE
                  NEXT SENTENCE.
```

Nested IF Statements

One of the more confusing uses of the IF statement involves placing additional IF statements following the THEN and/or ELSE portions. Our now familiar payroll variable, HOURS-WORKED, serves as an excellent example. HOURS-WORKED must be numeric and must also contain a value in the range of 1 to 40 to be declared free of error. One possible way to construct an IF statement to verify the content of this field is shown in figure 13.15. This may appear to be somewhat confusing. Note that the first condition inquires whether or not HOURS-WORKED is numeric. If it is, it makes sense to see if the numeric digits are in the range of 1 to 40. If HOURS-WORKED is not numeric, why bother checking to see if the digits are in the numeric range of 1 to 40?

If HOURS-WORKED is numeric, the THEN portion of the IF statement is entered. Here we encounter another IF statement. This second IF statement checks to see if our numeric value in HOURS-WORKED is in the range of 1 to 40. If it is, this field has passed the edit test and control passes to NEXT SENTENCE, which, in turn, causes the termination of the entire IF statement. If, on the other hand, HOURS-WORKED contains the number 81, control passes to the ELSE portion of the imbedded IF statement. Here an error message is printed, a bi-stable device is set, and so on.

What you may be wondering, however, is what happens to control if the condition in the first, or outermost, IF statement is false. When this occurs, control passes to the ELSE portion of this IF statement. Does the ELSE and its three related COBOL instructions belong to the outermost IF, the innermost IF, or possibly both? The answer is that the ELSE belongs to the innermost IF statement. The outermost IF statement does not have an ELSE. As a result, if the field is not numeric, control leaves the IF statement altogether, and no error message is ever printed. Thus our COBOL segment in figure 13.15 has a logic error. The error message is only printed if the field is numeric but contains a value outside of the range of 1 to 40. If the field is not numeric, no error message is printed.

```
           IF HOURS-WORKED NUMERIC
               THEN
                   IF HOURS-WORKED > "00" AND
                      HOURS-WORKED < "41"
                       THEN
                           NEXT SENTENCE
                       ELSE
                           DISPLAY ERROR-MESSAGE-3
                           MOVE "YES" TO TRANSACTION-IN-ERROR
                           ADD 1 TO FIELDS-FOUND-IN-ERROR.
```

Figure 13.15
Nested IF statements

Figure 13.16
Overcoming problem with nested IF statements

```
IF HOURS-WORKED NUMERIC
    THEN
        IF HOURS-WORKED > "00" AND
           HOURS-WORKED < "41"
            THEN
                NEXT SENTENCE
            ELSE
                DISPLAY ERROR-MESSAGE-3
                MOVE "YES" TO TRANSACTION-IN-ERROR
                ADD 1 TO FIELDS-FOUND-IN-ERROR
    ELSE
        DISPLAY ERROR-MESSAGE-3
        MOVE "YES" TO TRANSACTION-IN-ERROR
        ADD 1 TO FIELDS-FOUND-IN-ERROR.
```

Figure 13.17
Nested IF restated in simpler form

```
IF HOURS-WORKED NOT NUMERIC          OR
   HOURS-WORKED = "00"               OR
   HOURS-WORKED > "40"
    THEN
        DISPLAY ERROR-MESSAGE-3
        MOVE "YES" TO TRANSACTION-IN-ERROR
        ADD 1 TO FIELDS-FOUND-IN-ERROR.
```

Nested IF Statements
A device placing IF statements inside other IF statements that is used to expose a field to multiple tests.

The rule to remember in *nested* IF *statements* is that an ELSE belongs to the most recent IF statement. Since the ELSE immediately follows the innermost IF statement, it belongs to the innermost IF statement, leaving the outermost IF statement without an ELSE. To correct this problem the somewhat unnatural alternative in figure 13.16 is required. The program segment in figure 13.16 is unusual because two identical ELSE portions are required. Each, however, serves a separate IF statement and can only be activated under different conditions.

As mentioned earlier, nested IF statements can cause problems if they are not properly constructed. The best way to avoid these problems is to not use them. Any IF statement that seems ideal for use in a nested manner can always be stated as a single IF with multiple conditions. Figure 13.17 restates the nested IF in simpler form than that which was shown in figure 13.16. All parts of this three-part condition are connected by an OR. For this condition to be true, and hence the field to be in error, only one of these conditions need be true. If the field is not numeric, an error exists. If the field contains a 00, an error exists. If the value is greater than 40, an error exists.

Compound IF Statements

Compound IF Statements
Another device for exposing a field to multiple tests. It can be structured to avoid difficulties in interpretation and consequent program problems.

The alternative to the nested IF statement involves multiple conditions that are connected by ANDs and ORs. Figure 13.17 illustrates a *compound* IF *statement.* While compound IF statements may be easier to follow than nested IF statements, they are, nevertheless, troublesome.

Assume we were constructing an edit program to examine inventory transactions. Each transaction contains a department code and an item supply code. Also, assume department 12 can only possess a supply code of Q or blank. What would the IF statement look like to find out whether department 12 possesses a valid supply code? Figure 13.18 contains several IF statements for this purpose.

Alternative one looks promising. It is, however, incorrect. You see, alternative one is not structured to find an error. Instead, it is designed to find a good combination of department 12 and supply codes. A combination of 12Q or 12 will produce a true outcome. Unfortunately, a combination of 13 will also be considered good. In fact, it doesn't matter what value department takes on as long as the supply code is a blank as a result of the precedence of logical connectives.

Alternative one

```
IF DEPARTMENT = "12" AND
   SUPPLY-CODE = "Q" OR
   SUPPLY-CODE = " "
    THEN
         we hope combination is good.
```

Alternative two

```
IF DEPARTMENT = "12" AND
   (SUPPLY-CODE = "Q" OR
   SUPPLY-CODE = " ")
    THEN
         we know the combination is
         good.
```

Alternative three

```
IF DEPARTMENT = "12" AND
   SUPPLY-CODE = "Q" OR " "
    THEN
         combination is also good.
```

Figure 13.18
Alternatives in compound conditions

Precedence of Connectives

A compound IF statement, in which all the conditions are connected by ORs, contains very little mystery. The logical OR stipulates that all it takes is a single condition to be true and the entire IF statement is true. If two of the conditions are true, the entire IF statement is also true, but it only requires one.

A compound IF statement featuring only ANDs is somewhat more demanding. In order for the entire IF statement to be true, all of the conditions must be true, without exception.

When ANDs and ORs are used together in the same IF statement, the rule to interpreting the IF statement is: ANDs are evaluated before ORs. Consequently, in the first alternative in figure 13.18, the conditions on either side of the AND will be evaluated first. The outcome of that evaluation will then use ORs with the last of the three conditions. Because the OR is evaluated last, and because an OR only requires that one condition be true if the entire IF is to be true, any value for department is acceptable as long as the supply code is a blank.

Correcting this problem involves the use of parentheses as shown in alternative two. Parentheses dictate the order in which the logical operators are evaluated. Now the OR is evaluated first. This means that for the OR to produce a true outcome, either the supply code is a Q or blank. If it is neither, the OR is false. If the OR is false the AND is also false, because ANDs permit no falsehoods if the outcome is to be true.

A second way to eliminate the problem in alternative one involves expressing part of the multi-conditional in an abbreviated form. Note that alternative three shows that the supply code must be either a Q or blank. The AND does not force the OR to be processed as it was in alternative one.

Use of Condition Names and Variables

The problem with all of the alternatives in figure 13.18 is that they were designed to find a good combination of department and supply code. How does the IF statement look if we want to find a bad combination instead?

Figure 13.19
Use of condition names/variables in compound IFs

Alternative one

```
       02 DEPARTMENT              PICTURE IS X(02).
       02 SUPPLY-CODE             PICTURE IS X(01).

       IF DEPARTMENT = 12         AND
          SUPPLY-CODE NOT = "Q"   AND
          SUPPLY-CODE NOT = " "
              THEN
                 error detected.
```

Alternative two

```
       02 DEPARTMENT              PICTURE IS X(02).
       02 SUPPLY-CODE             PICTURE IS X(01).
          88 VALID-SUPPLY-CODE    VALUE "Q", " ".

       IF DEPARTMENT = "12"
              AND
          NOT VALID-SUPPLY-CODE
              THEN
                 error detected.
```

Examine alternative one in figure 13.19. If SUPPLY-CODE is not equal to Q and not equal to a space, then it is incorrect. A value of V, for instance, is not equal to Q, AND is not equal to a space either. Thus the entire condition is true, and an error has been found.

Some might find it difficult to interpret the IF statement shown in alternative one of figure 13.19. The combination of not equals and AND do require some careful thought. Consequently, we have used the condition variable in alternative two.

The second alternative in figure 13.19 is much easier to comprehend. We know the condition VALID-SUPPLY-CODE can only occur if the variable SUPPLY-CODE contains either a Q or a space. If it contains neither, the condition does not exist. Hence, if the department is 12, and the condition does not exist, we have an error. In general, the use of a condition variable in compound IF statements can greatly increase the reader's comprehension.

Summary

The COBOL IF statement lies at the heart of implementing edit programs. This statement can be used to examine the contents of each field in every transaction processed. It can also be used to find out the value of bi-stable devices.

Bi-stable devices are needed in the edit program to determine if the record currently being processed contains one or more fields in error. These devices are regular COBOL variables defined as 01 level items in our program's DATA DIVISION. The attributes of bi-stable devices are relatively unimportant to the execution of the program by the computer. These devices can be either nonnumeric variables (PICTURE IS X) or numeric variables (PICTURE IS 9)—either of which can work quite well.

Nonnumeric variable uses of bi-stable devices offer the advantage of improved reading comprehension. The two values of a bi-stable variable can be YES or NO (or their abbreviations Y or N, respectively). Clearly, YES, or Y, indicates that an error has been located. NO, or N, specifies the current record is free of fields in error.

Numeric values suffer from the lack of clarity of YES or NO. Here the values are likely to be zero or one. It may be difficult to remember whether zero means NO, or YES. To overcome the limitations of numeric implementations of bi-stable devices, COBOL offers symbolic devices known as condition names.

Condition names are not traditional variables, although they certainly look like them. They are not allocated primary memory space in the record structures where they are defined. Condition names are always defined as 88-level items in the B-Margins of the DATA DIVISION. Programs using condition names can make use of IF statements that are more descriptive and more meaningful to the reader than IF statements written in less descriptive form.

COBOL also offers class tests to determine if fields contain NUMERIC or ALPHABETIC values. These tests are valuable because constructing numeric range tests for multiple-digit fields would be an awesome task. Essentially, to verify the fact that a supposedly numeric field contains numeric digits would require that each and every position within that field be checked. The resulting IF statement would be quite long.

Some fields in an input transaction require multiple edit criteria. Not only, for instance, should a field be numeric, but it must contain only a specific range of numeric values. Not all numeric values are acceptable. An interesting device for implementing multiple edit criteria in an edit program is the nested IF statement.

While they might at first appear intriguing, nested IF statements can also prove to be quite troublesome for the beginner. Nested statements prove difficult to read and interpret. As an earlier example illustrated, if structured incorrectly the program's execution can be perfect in one instance and yet completely incorrect in another. The option of multiple conditions in a single IF statement seems to offer all of the advantages of nested IF statements with less likelihood of error.

Unfortunately, given enough ANDs and ORs, even a compound conditional can become quite difficult to handle. Mixtures of ANDs and ORs can produce an outcome entirely different from what is desired due to the precedence of AND over OR. While parentheses can force interpretation in the manner desired by the programmer, the use of negated operators makes for a potentially troublesome device.

To overcome the complexities of compound conditions, the COBOL programmer can rely upon the use of condition variables. These variables make it possible to condense a lengthy list of conditions into a single, easy to understand statement.

Questions for Review

1. Why are bi-stable devices necessary in edit programs?
2. How does the iterative instruction in the MAIN-CONTROL SECTION of an edit program (or report program) treat the input record structure as a bi-stable device?
3. What is a condition name? Of what value is it?
4. Why is it impossible to do much more than verify the presence of nonblank data in name fields?
5. Why can't multiple digit numeric fields be checked with a single range check?
6. Why are supposedly numeric fields defined as nonnumeric fields in edit programs but not as such in report programs?
7. What is a nested IF statement and why is it so difficult to use?
8. What is a compound IF statement? Why are they required in edit programs?
9. Examine the IF statement in figure 13.9. What, if anything, is wrong?

Problems

1. What, if anything, is wrong with the following COBOL statements? For the first two statements, assume TRANSACTION-IN-ERROR is a nonnumeric variable. In the last, assume it is a numeric variable.

```
MOVE YES TO TRANSACTION-IN-ERROR.
MOVE TRANSACTION-IN-ERROR TO YES.
MOVE "0" TO TRANSACTION-IN-ERROR.
```

2. Examine the following segments from a COBOL edit program very carefully. Does the example illustrate the proper use of condition names? If not, why not?

```
01   TRANSACTION-IN-ERROR        PICTURE IS 9(01) VALUE 0.
     88 ERROR-FOUND              VALUE 1.
     88 NO-ERROR-FOUND           VALUE 2.
IF TRANSACTION-IN-ERROR = ERROR-FOUND
     THEN
         record contained at least one bad
         field.
```

3. Assume that an input record contains a field description like:

```
02 HOURLY-WAGE        PICTURE IS 9V99.
```

Redefine this variable and write the COBOL statements to verify that each of its three digits is numeric. You may not use the NUMERIC class test.

4. What, if anything, is wrong with the use and intent of the following nested IF statement?

```
IF HOURS-WORKED NUMERIC
    THEN
        IF HOURS-WORKED > "00"
           HOURS-WORKED < "41"
           THEN
               NEXT SENTENCE.
    ELSE
        DISPLAY ERROR-MESSAGE-3
        MOVE "YES" TO RECORD-IN-ERROR
        ADD 1 TO FIELDS-FOUND-IN-ERROR.
```

5. Examine the following segments from an edit program. Typically, messages are not printed when a field is shown to be free of error. However, in this specific instance, assume an appropriate message will be printed at the conclusion of the editing process for a field named HOURS-WORKED whether the field was in error or not. What, if anything, is wrong with this IF statement?

```
IF HOURS-WORKED = NUMERIC
    THEN
        IF HOURS-WORKED > "00"
            THEN
                IF HOURS-WORKED < "41"
                    THEN
                        NEXT SENTENCE
            ELSE
                DISPLAY ERROR-MESSAGE-3
                MOVE "YES" TO RECORD-IN-ERROR
                ADD 1 TO FIELDS-FOUND-IN-ERROR
    ELSE
        DISPLAY GOOD-MESSAGE.
```

6. In a single compound IF statement, you are to examine two fields from a single transaction. One field contains the two-digit abbreviation for the month of the year. The second field contains a two-digit day of the month. Your IF statement is to verify that the combination of month and day is legitimate. Ignore the issue of leap year.
7. Shorten the IF statement produced in exercise 6 using condition variables.
8. An edit program is to examine a field named *DEPARTMENT* that belongs to the input record description. DEPARTMENT must be numeric and it can contain only department numbers: 01, 03, 05, 08, and 12. Examine the following:

```
02 DEPARTMENT                PICTURE IS X(02).
   88 VALID-DEPTS            VALUE 01, 03, 05,
                                   08, 12.
IF NOT VALID-DEPTS    OR
   DEPARTMENT NOT NUMERIC
    THEN
         an error has been detected.
```

What, if anything, is wrong? Why? Were both conditions required? If not, why not?

Programming Assignments

1. Pseudonym Systems, an Arizona data processing services firm, has been retained by the Senior Citizens Association of the Greater Southwest (SCAGs) to apply computer-based technology to the association's membership record keeping system. Until now, all of SCAGs members applied for membership, enclosed proper payment with their application, and subsequently received a hand-prepared membership card. In general, all of the record keeping tasks associated with maintaining an accurate list of SCAG members was performed entirely without the benefit of any automation.

   Pseudonym Systems has designed an automated membership record keeping and reporting system for use by SCAGs. Newly designed application forms are to be found accompanying all of the advertisements placed by SCAGs in the print media. These applications are completed by each prospective member and forwarded to the SCAGs office. Once received by SCAGs, these forms are transcribed onto a machine-readable medium. However, before each new member's record is added to the SCAGs membership data base, it is examined to verify that the data contained is without error. Each of the membership records conforms to the following layout:

Record Position	*Data Type*	*Variable*
01–19	AN	Member's name
20–35	AN	Street address
36–45	AN	City name
46–47	AN	State abbreviation code
48–56	N	Zipcode
57–58	N	Month membership expires
59–60	N	Year membership expires
61–66	N	Member's date of birth (MMDDYY)
67–80	—	Unused

An edit report like that shown in figure 13.20 is to be produced.

Figure 13.20
Senior Citizens Association
edit report

PROGRAMMER ____________

PROBLEM ____________

DATE ____________

REPORT LAYOUT FORM

PAGE 1 OF 2

```
REPORT MS-E-01                                    SENIOR CITIZENS ASSOCIATION                                   02-25-81
                                                             OF THE
                                                       GREATER SOUTHWEST

                                                       MEMBERSHIP SYSTEM
                                                  EDIT OF APPLICATION DATA

                  .........1.........2.........3.........4.........5.........6.........7.........8
TRANSACTION 001 ->LARMAN A WILSON     1400 WILSON *045ARLINGTON VA22191000008810211 16

TRANSACTION 002 ->                                      ALEX                 08
                      ** ERROR ** NAME MISSING
                      ** ERROR ** ADDRESS MISSING
                      ** ERROR ** STATE MISSING
                      ** ERROR ** ZIPCODE MISSING/INVALID
                      ** ERROR ** YEAR MISSING/INVALID
                      ** ERROR ** DATE OF BIRTH MISSING/INVALID

TRANSACTION 003 ->HUGH BOYNTON        RFD 2             WACO       TX54351000008810419 18

                  .........1.........2.........3.........4.........5.........6.........7.........8
                                                   PAGE-001
```

PROGRAMMER ____________

PROBLEM ____________

DATE ____________

REPORT LAYOUT FORM

PAGE 2 OF 2

```
REPORT MS-E-01                                  SENIOR CITIZENS ASSOCIATION                                             02-25-81
                                                          OF THE
                                                    GREATER SOUTHWEST

                                                   MEMBERSHIP SYSTEM
                                              EDIT OF APPLICATION DATA

                                                        SUMMARY

                                    TRANSACTIONS READ  XXX

                                             IN ERROR  XXX

                                         PCT IN ERROR  XXX%

                                                      PAGE-XXX
```

2. Eglin Auto Distributors serves as the wholesale distributor of remanufactured autoparts for imported and classic automobiles in the tri-state area of Pennsylvania, Maryland, and West Virginia. The volume of auto parts shipped to dealers and service stations has made it difficult to manually prepare sales invoices and statements of account in a timely manner. As a result, Eglin is experiencing a large cash flow problem which has necessitated the reliance on expensive lines of credit from its banks.

   The local data processing firm used for payroll processing by Eglin is willing to develop and implement an inventory/sales system for Eglin. The first phase of Eglin's proposed new system requires that all sales invoices continue to be hand prepared and then transcribed onto a machine-readable medium.

   Since transcription is far from perfect, each transcribed record must be examined to verify error-free data before the record is further processed by the new data processing system. The layout of each record follows:

Record Position	*Data Type*	*Variable*
01–06	N	Invoice number
07–12	N	Customer number
13–15	N	Quantity of item purchased
16–26	AN	Inventory number
27–28	AN	Unit of measure (i.e., BX, EA C, K)
29–34	N	Unit price
35–36	N	Sales tax rate (V99)
37–42	N	Date of order (YYMMDD)
43–48	N	Date of shipping (YYMMDD)
49–80	—	Unused

   The edit report to be produced by this program can be found in figure 13.21.

Figure 13.21
Eglin Auto parts edit report

PROGRAMMER ______

PROBLEM ______

DATE ______

REPORT LAYOUT FORM

PAGE ____ OF ____

```
                                                   EGLIN AUTO DISTRIBUTORS                             PAGE-001
                                                    INVENTORY MGMT SYSTEM
                                                   EDIT OF SALES INVOICES
                                                             FOR
                                                          05-01-80

                  ..........1.........2.........3.........4.........5.........6.........7.........8
      INVOICE     1:
                        ** ERROR ** MISSING/INVALID INVOICE NUMBER (1-6)
                        ** ERROR ** MISSING/INVALID CUSTOMER NUMBER (7-12)
                        ** ERROR ** MISSING/INVALID QUANTITY (13-15)
                        ** ERROR ** MISSING INVENTORY NUMBER (16-26)
                        ** ERROR ** MISSING/INVALID UNIT OF MEASURE (27-28)
                        ** ERROR ** MISSING/INVALID PRICE (29-34)
                        ** ERROR ** MISSING/INVALID SALES TAX (35-36)
                        ** ERROR ** MISSING/INVALID DATE OF ORDER (37-42)
                        ** ERROR ** MISSING/INVALID DATE OF SHIPPING (43-48)

      INVOICE     2:000139000019003 1A42200-Z-QEAD134990681031 7810402

      INVOICE     3:000J4300012300 12C66430-A-QBX009999AA81031 88A0402
                        ** ERROR ** MISSING/INVALID INVOICE NUMBER (1-6)
                        ** ERROR ** MISSING/INVALID SALES TAX (35-36)
                        ** ERROR ** MISSING/INVALID DATE OF SHIPPING (43-48)

                  ..........1.........2.........3.........4.........5.........6.........7.........8
```

14 Identification Division Revisited

Chapter Outline

Since the beginning of this text you have been shielded from many facets of the COBOL programming language. The reason for such an approach is that many of the statements in COBOL are unnecessary to construct a program. They are unnecessary in the sense that the COBOL compiler will accept these incomplete COBOL programs and create a machine-executable program without using all of the statements that the language features. By not presenting these unnecessary statements, the amount of COBOL that must be read, understood, and mastered at the outset is reduced rather dramatically. This approach is not without its problems, however.

Students learning any programming language tend to avoid the use of statements identified as less than absolutely critical. Program comments serve as an excellent example. Program comments are not absolutely critical to the translation of COBOL source programs by the COBOL compiler. It is possible to construct COBOL programs without any comments whatsoever, and these programs will be compiled successfully. That is, the compiler will not issue any warning or error messages like those shown in figure 14.1. While many statements are not absolutely critical to the translation of COBOL programs by the COBOL compiler, they are necessary to the programming process.

```
**ERROR** THIS PROGRAM DOES NOT CONTAIN
ANY COMMENTS IN THE PROCEDURE DIVISION.
PROGRAMS WITHOUT COMMENTS ARE NOT ACCEP-
TABLE IN THE INDUSTRY—THEY IMPAIR THE
INITIAL DEVELOPMENT AND SUBSEQUENT MAIN-
TENANCE EFFORTS.

                    or

**ERROR** THERE IS (ARE) ONE OR MORE
SECTIONS IN THE PROCEDURE DIVISION THAT
ARE MISSING PREFATORY COMMENTS. SEC-
TIONS NOT COMMENTED DO NOT CLEARLY STATE
THE PROGRAM'S FUNCTIONS COMPLETELY.

                or better yet

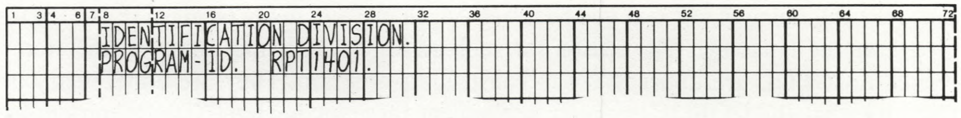
**ERROR** AT LEAST ONE PARAGRAPH IN THE
PROCEDURE DIVISION LACKS PREFATORY COM-
MENTS. EVEN THOUGH THE CONTAINING SEC-
TION IS COMMENTED THE SUBORDINATE PARA-
GRAPH IS NOT. HOW IS ONE EXPECTED TO
UNDERSTAND HOW PARAGRAPH CONTRIBUTES TO
THE FUNCTION(S) PERFORMED BY THIS SEC-
TION?

                and finally

**ERROR** PROGRAM COMPILATION ABORT-
ED** BASED ON THE MISSING COMMENTS,
THERE IS ABSOLUTELY NO NEED TO CONTINUE
THE COMPILATION AND POSSIBLE EXECUTION
OF THIS PROGRAM. THIS COMPILER REFUSES
TO PARTAKE IN SUCH A "CRIME".
```

Figure 14.1
Wishful thinking by the author

Figure 14.2
Required entry in
IDENTIFICATION DIVISION



Basic Structure of the Identification Division

The IDENTIFICATION DIVISION is comprised of two types of statements. Some of the statements are absolutely required if the COBOL compiler is to translate the source program, and some are not nearly so critical. The only two entries that have been discussed thus far are shown in figure 14.2. The first entry is required because it marks the start of the division named IDENTIFICATION. Remember, every COBOL program must possess four divisions—IDENTIFICATION, ENVIRONMENT, DATA, and PROCEDURE. Omitting one of these divisions causes the COBOL compiler to reject the source program being translated. Consequently, the first entry must appear in each COBOL program.

Statement Format

```
8
↓
IDENTIFICATION DIVISION.
```

The second of the two entries shown in figure 14.2, PROGRAM-ID, is also required. This particular entry identifies the name that the programmer has chosen to uniquely identify this program. In the example shown in figure 14.2, the program's name is RPT1401.

Statement Format

```
8
↓
PROGRAM-ID. programname.
```

Absence of Sections and Paragraphs

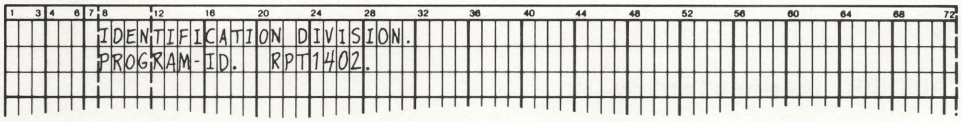
The PROCEDURE DIVISION is made up of sections. Each section performs a function that is critical to the overall task assigned to the program. Sections, in turn, are composed of paragraphs. Paragraphs embody the subfunctions that, when grouped together, give the containing section the function for which it was designed. Paragraphs are constructed of COBOL statements of action, such as arithmetic instructions, input/output instructions, and so on.

The use of sections, paragraphs, and entries (statements) characterizes three of the four divisions in a COBOL program. Those divisions are the ENVIRONMENT DIVISION, DATA DIVISION, and PROCEDURE DIVISION. For reasons clear only to the original designers of the COBOL language, the IDENTIFICATION DIVISION departs from this structure. The IDENTIFICATION DIVISION is not composed of sections. Instead, this particular division contains only paragraphs and entries for those paragraphs.

The Use of Margins

All of the paragraphs in the IDENTIFICATION DIVISION must begin somewhere in the A-Margin. This margin starts in column 8 of the source statement and ends in column 11. Column 12 marks the start of the B-Margin. As a matter of consistency, all of the statements in the IDENTIFICATION DIVISION shown thus far have been started in the same column. We could have chosen to use indentation, however. Figure 14.3 illustrates a possible use of indentation. In this instance, indentation shows ownership and hierarchy more clearly. Note, however, that even though indentation has been used, both entries begin in the A-Margin. The first might begin in column 8, and the second might begin in column 10.

Alternative one



Alternative two

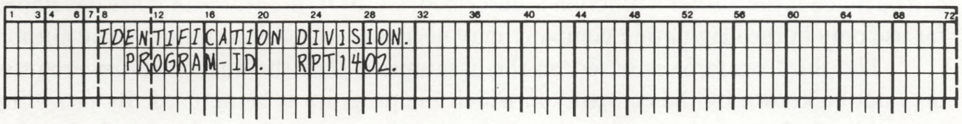
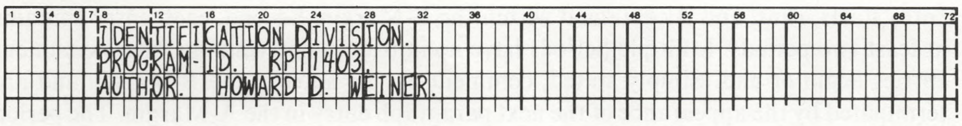
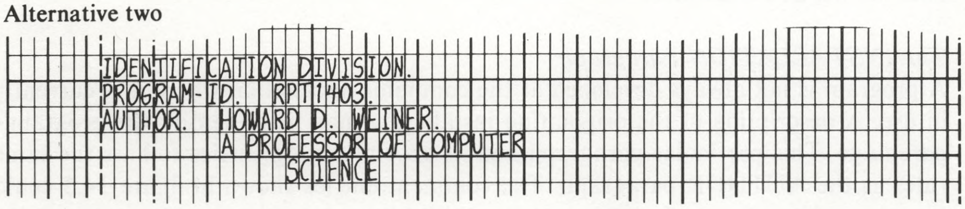


Figure 14.3
Indentation in the IDENTIFICATION DIVISION

Alternative one



Alternative two



Alternative three

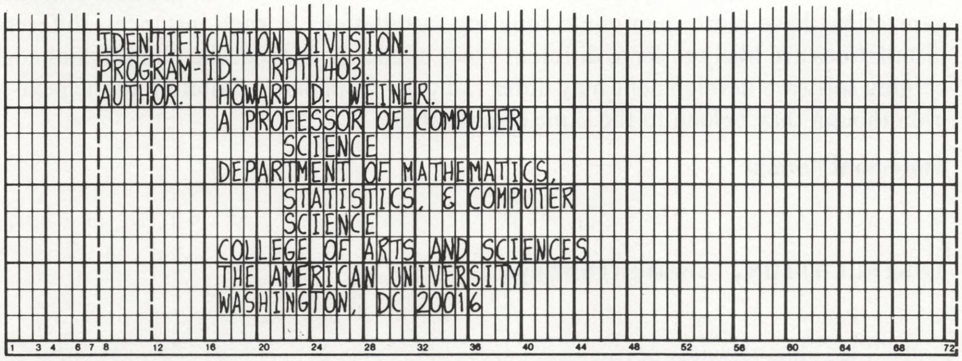


Figure 14.4
Examples of the AUTHOR paragraph entry

Additional Paragraphs for the Identification Division

The IDENTIFICATION DIVISION contains six other paragraphs that can and should appear in each and every COBOL program. Each of these paragraphs is followed by one or more passive comment entries, that can be as short or as lengthy as the programmer chooses to make them. Each comment entry must be wholly contained in the B-Margin. If the comment requires more than one source statement, the comment can continue on the next source statement as long as the continuation starts in the B-Margin. A description of each paragraph follows.

The AUTHOR Paragraph

This entry identifies the programmer, or possibly programmers, who contributed to the design, testing, and implementation of the COBOL program. Examine the samples shown in figure 14.4. Each example shows AUTHOR paragraph entries of varying degrees of complexity. In the first example the only purpose is to identify who is responsible for the COBOL program in which this entry is contained. This form of AUTHOR paragraph will suffice if everyone who reads this program knows who the author is. This is not likely to be the case in all computing installations, however.

Statement Format

```
8
↓
AUTHOR. commententry...
```

The second example is based on the assumption that a more precise identification can be made if the author's function is described. COBOL programs that are developed for use only in the originating company may need no further description of the author.

The third example is based on the assumption that the author of the program may not be known by the company in which the program is being used. By providing several lines of comment entries, the reader can determine who the author is, in what capacity the author operates, and for what organization or firm the author works.

Note that all three examples have several things in common. First, each comment entry starts, runs, and is contained within the B-Margin. Second, each entry contains at least two periods. Multiple periods have no impact in determining where comment entries begin or end. Simply stated, comment entries start in the B-Margin and are terminated by the appearance of the next paragraph entry in the A-Margin. The period does not act as a delimiter.

The INSTALLATION Paragraph

A program is not always developed at the computer installation where it is currently in use. A description of the originating installation may help identify the program's parentage. This particular paragraph and its accompanying entries follows the same rules as the AUTHOR paragraph. The comment entries shown in figure 14.5 could have been fewer in number. Nevertheless, complete identification of the computing installation used might require more than one line of qualification, because some organizations have more than one computing installation. In such instances, different installations are likely to serve different types of clients. Indeed, the philosophies concerning software development may differ radically from one installation to the next—even though they belong to the same organization.

Statement Format

```
8
↓
INSTALLATION. commententry...
```

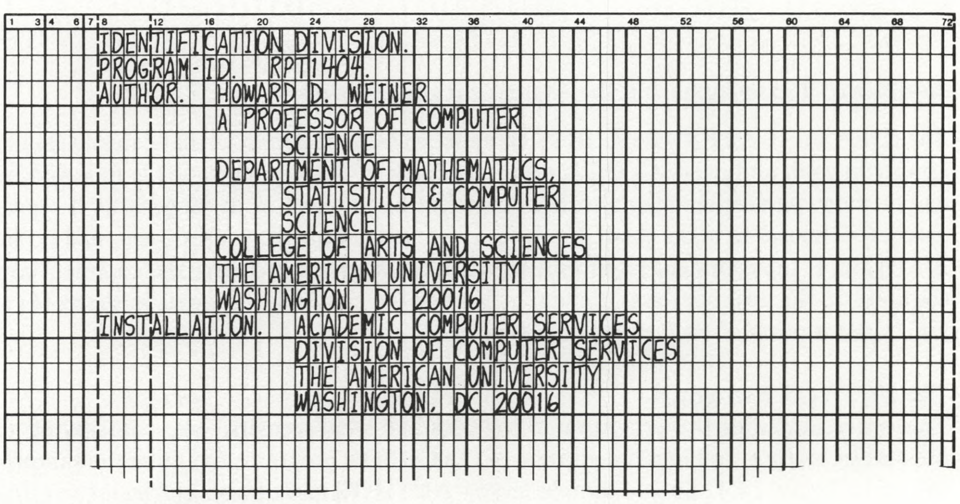


Figure 14.5
Use of the INSTALLATION paragraph entry

The DATE-WRITTEN Paragraph
An additional piece of information identifying a COBOL program is when the program was first written. This industry has experienced tremendous changes in programming techniques and technologies in the past decade. By determining when the program was written, the reader can infer what design and programming concepts were being used in the originating organization at that time. Information such as this can be very valuable to the reader of the program.[1]

The DATE-WRITTEN paragraph is shown in figure 14.6. Note that this paragraph has only one comment entry following it. This does not mean that only one comment entry is possible. In fact, as many comment entries can appear here as the program's author chooses to use.

Statement Format

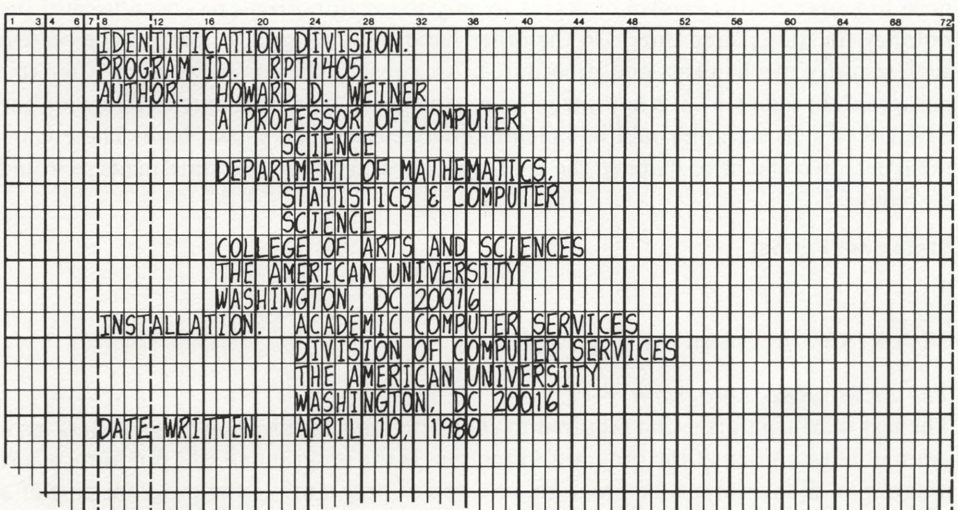
```
8
↓
DATE-WRITTEN. commententry...
```

One frequent error made by inexperienced COBOL programmers is the omission of the hyphen between the words DATE and WRITTEN. Remember, the COBOL compiler is very unforgiving when it comes to incorrect spelling. For this reason, you must always remember to place the hyphen between the two words in this paragraph name.

The DATE-COMPILED Paragraph
The COBOL programmer using this particular paragraph in a COBOL program may choose to supply the date the program was last compiled by the COBOL compiler. However, when the COBOL compiler encounters the DATE-COMPILED paragraph in the COBOL program, it supplies the current date automatically. To have the compiler supply the date of compilation, the programmer need only supply the paragraph. The comment entry or entries that follow will automatically be replaced with the date the program was compiled by the COBOL compiler.

Statement Format

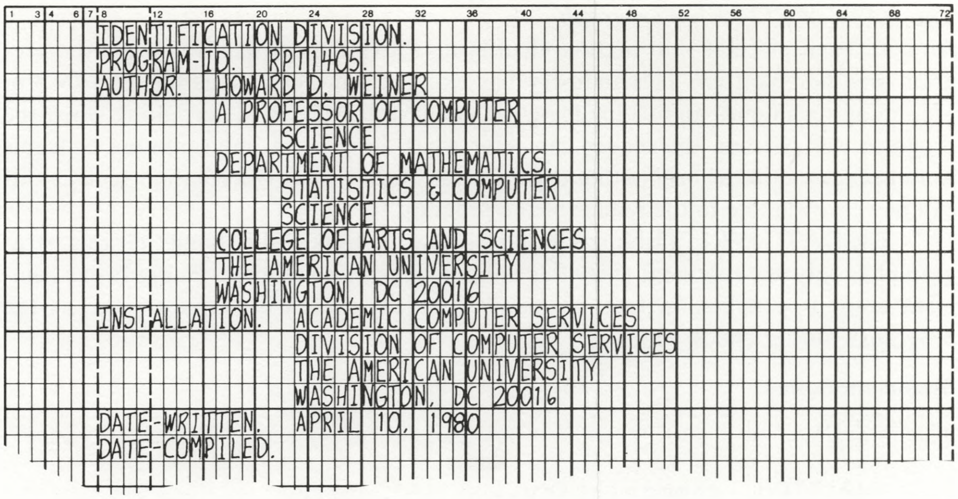
```
8
↓
DATE-COMPILED. currentdate
```



```
IDENTIFICATION DIVISION.
PROGRAM-ID.  RPT1405.
AUTHOR.  HOWARD D. WEINER
         A PROFESSOR OF COMPUTER
               SCIENCE
         DEPARTMENT OF MATHEMATICS,
               STATISTICS & COMPUTER
               SCIENCE
         COLLEGE OF ARTS AND SCIENCES
         THE AMERICAN UNIVERSITY
         WASHINGTON, DC 20016
INSTALLATION.  ACADEMIC COMPUTER SERVICES
               DIVISION OF COMPUTER SERVICES
               THE AMERICAN UNIVERSITY
               WASHINGTON, DC 20016
DATE-WRITTEN.  APRIL 10, 1980
```

Figure 14.6
Use of the DATE-WRITTEN paragraph entry

Figure 14.7
A more completely specified IDENTIFICATION DIVISION



```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.  RPT1405.
       AUTHOR.   HOWARD D. WEINER
                 A PROFESSOR OF COMPUTER
                       SCIENCE
                 DEPARTMENT OF MATHEMATICS,
                       STATISTICS & COMPUTER
                       SCIENCE
                 COLLEGE OF ARTS AND SCIENCES
                 THE AMERICAN UNIVERSITY
                 WASHINGTON, DC 20016
       INSTALLATION.   ACADEMIC COMPUTER SERVICES
                       DIVISION OF COMPUTER SERVICES
                       THE AMERICAN UNIVERSITY
                       WASHINGTON, DC 20016
       DATE-WRITTEN.   APRIL 10, 1980
       DATE-COMPILED.
```

Many COBOL students have attempted to alter the date of compilation to convince the course instructor that a programming assignment was completed within the alloted time. After all, if an assignment due on May 10, 1980 did not work properly until May 11, the student might be tempted to supply a comment entry after the DATE-COMPILED paragraph like:

```
DATE-COMPILED. MAY 10, 1980
```

Unfortunately, the COBOL compiler, when encountering this paragraph, will replace the date of May 10 with the actual date of compilation so that the entry in the source listing will appear as:

```
DATE-COMPILED. MAY 11, 1980
```

COBOL is an unwilling accomplice.

The IDENTIFICATION DIVISION now appears as shown in figure 14.7.

The SECURITY Paragraph

Many organizations view their programs and files as valuable corporate resources. To safeguard these resources, many companies have established one or more levels of security classification. People not authorized at a particular level of security are unable to access certain data and programs. The SECURITY paragraph permits the programmer to include the level of security desired. Such use of security, establishing the need to know and the control of access to only those authorized, has been used in the military for quite some time. On the other hand, many organizations do not possess such security mechanisms and procedures. As a result, the comment entry after this paragraph in figure 14.8 is not uncommon. Again, the comment entries following the SECURITY paragraph could be much more extensive than shown here. These comment entries could have spanned the B-Margins of as many source statements as would be required.

Statement Format

```
8
↓
SECURITY. commententry...
```

The REMARKS Paragraph

The last item of information required to complete the identification of a COBOL program is a description of the program's function. Is the program an edit program, a sort program, a file maintenance program, or a report program? Is the program part of a subsystem concerned with corporate payroll or inventory? What are the program's inputs? Who supplies the inputs? To whom do the printed results go? As you can see, this paragraph, by its very nature, requires a significant number of comment entries each of which span the B-Margins of many source statements. An excellent example can be found in figure 14.9.

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.  RPT1405.
       AUTHOR.    HOWARD D. WEINER
                  A PROFESSOR OF COMPUTER
                         SCIENCE
                  DEPARTMENT OF MATHEMATICS
                         STATISTICS & COMPUTER
                         SCIENCE
                  COLLEGE OF ARTS AND SCIENCES
                  THE AMERICAN UNIVERSITY
                  WASHINGTON, DC 20016
       INSTALLATION.   ACADEMIC COMPUTER SERVICES
                       DIVISION OF COMPUTER SERVICES
                       THE AMERICAN UNIVERSITY
                       WASHINGTON, DC 20016
       DATE-WRITTEN.   APRIL 10, 1980
       DATE-COMPILED.
       SECURITY.   NONE WHATSOEVER.
```

Figure 14.8
Common use of the SECURITY paragraph entry

Page 1 of 2

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.  RPT1405.
       AUTHOR.    HOWARD D. WEINER
                  A PROFESSOR OF COMPUTER
                         SCIENCE
                  DEPARTMENT OF MATHEMATICS,
                         STATISTICS & COMPUTER
                         SCIENCE
                  COLLEGE OF ARTS AND SCIENCES
                  THE AMERICAN UNIVERSITY
                  WASHINGTON, DC 20016
       INSTALLATION.   ACADEMIC COMPUTER SERVICES
                       DIVISION OF COMPUTER SERVICES
                       THE AMERICAN UNIVERSITY
                       WASHINGTON, DC 20016
       DATE-WRITTEN.   APRIL 10, 1980
       DATE-COMPILED.
       SECURITY.   NONE WHATSOEVER.
       REMARKS.   THIS IS A REPORT PROGRAM THAT DETAILS
               PAYROLL COSTS FOR EACH DIVISION AND
               REGION IN THE COMPANY.  EVERY EMPLOYEE
               WILL HAVE HIS SALARY LISTED FOR THE PAY PERIOD
               JUST COMPLETED.  INPUT TO THE PROGRAM COMES
               FROM THE FILE OF EDITED TIMECARDS.  THE REPORT
```

Page 2 of 2

```
           IS DISSEMINATED TO THE APPROPRIATE LEVELS OF
           OPERATING MANAGEMENT THROUGHOUT EACH DIVISION,
           REGION, AND CENTRAL OFFICE.
```

Figure 14.9
Completed IDENTIFICATION DIVISION

Statement Format

```
8
↓
REMARKS. commententry...
```

This particular paragraph entry was removed from the formal definition of the COBOL language with the acceptance of the 1974 standard (COBOL74). This deletion is unfortunate. However, many, many COBOL programs have been written using this paragraph entry. Indeed, even though the REMARKS paragraph is no longer a part of COBOL, many programmers continue to use it.

Summary

The IDENTIFICATION DIVISION includes many passive entries containing information concerning the development of a COBOL program. Many of these paragraphs and comment entries have not been introduced in this text until now because it was not essential to complicate the learning process with them. No matter how highly desireable it is that they be included, they can be omitted by beginning COBOL programmers. In this manner much of COBOL has been hidden. It is now appropriate that these paragraphs and comment entries be made a part of the COBOL programs we develop.

Each of the paragraphs in the IDENTIFICATION DIVISION must start in the A-Margin of the COBOL source statement. Their corresponding comment entries can be as short or as long as the programmer chooses. A comment entry begins in the B-Margin after the prefacing paragraph entry ends. The comment entry ends when a new paragraph entry is encountered in the A-Margin. Comment entries may contain as many marks of punctutation as the programmer would like, because punctuation does not mark the end of a comment entry.

Only the PROGRAM-ID paragraph is required in each COBOL program. The remaining paragraph entries are optional, technically speaking, although all should appear for the reasons discussed earlier. When used, these paragraph entries must appear in the order in which they were introduced. For example, if SECURITY is used, it must follow all of the paragraphs but REMARKS; if AUTHOR is used, it must immediately follow PROGRAM-ID.

Notes

1. It is quite possible that programs written before the advent of structured- or modular-programming techniques may include unconditional transfers of control (GO TO) too frequently or may use them improperly. Likewise, such programs may exhibit little in the way of module definition and invocation.

Questions for Review

1. It is certainly far easier to construct COBOL programs that do not contain the many paragraph and comment entries discussed in this chapter. What set of circumstances could make the program interpretation a far more difficult task without a complete IDENTIFICATION DIVISION?

2. In what respect does the IDENTIFICATION DIVISION differ from the other divisions in a COBOL program?

3. What, if anything, is wrong with the following IDENTIFICATION DIVISION from a COBOL program?

```
IDENTIFICATION DIVISION.
     INSTALLATION   ABCE COMPUTER SYSTEMS, INC.
     DATE-COMPILED.
     DATE-WRITTEN   MAY 25, 1980
     REMARKS.  THIS IS AN EDIT PROGRAM FOR A
               PAYROLL SYSTEM THAT WAS DEVELOPED
               FOR A SMALL MANUFACTURING CONCERN.
     SECURITY.  NONE.
```

4. Read the following statement very carefully. Is it a correct summary of the rules for writing comment entries in the IDENTIFICATION DIVISION? If not, why not?

   Comment entries can begin anywhere on any source statement following the last alphabetic character of the paragraph entry to which it belongs.

   Comment entries that span more than one source statement must contain a hyphen in column 7 of the continued source statement.

   Periods must appear only at the end of the last comment entry belonging to a paragraph. Since periods terminate comment entries, the first appearance of a period causes the COBOL compiler to consider the comment entry ended even if there are additional comments following the period for the same paragraph.

Problems

1. Examine the complete COBOL program shown in figure 3.14. Construct a complete IDENTIFICATION DIVISION for that program.
2. Assume you are constructing the program in figure 3.14 on a contract basis for another firm. How would you modify the paragraph entries in exercise 1? Why?
3. What changes would have to be made to reflect the fact that a team of four programmers built this program as a group endeavor?

Programming Assignments

1. Revise the Senior Citizens edit report program from chapter 13 to include a fully specified IDENTIFICATION DIVISION.
2. Revise the Eglin Auto Parts edit report from chapter 13 to contain a fully specified IDENTIFICATION DIVISION.

15 Environment Division Revisited

Chapter Outline

Like the statements in the IDENTIFICATION DIVISION, much of the ENVIRONMENT DIVISION has been hidden as well. Unlike the IDENTIFICATION DIVISION, however, this particular division is faithful to the structure used in the PROCEDURE DIVISION. That is, it is a division composed of sections, which are composed of paragraphs, which in turn are composed of statements.

The ENVIRONMENT DIVISION contains only two sections. They are the CONFIGURATION SECTION and the INPUT-OUTPUT SECTION. The CONFIGURATION SECTION is the topic of this particular chapter. The INPUT-OUTPUT SECTION will be discussed in a following chapter.

Statement Format

```
8
↓
ENVIRONMENT DIVISION.
```

The Configuration Section

This section describes the computing environment in which the COBOL program is compiled and executed. This section, and its paragraph entries, is optional. As a result, our COBOL programs, which have not used this section and its contents, have compiled and executed successfully. Like many of the paragraphs in the IDENTIFICATION DIVISION, However, a great deal of helpful documentation can be contained in this section and its statements.

Statement Format

```
8
↓
CONFIGURATION SECTION.
```

If the paragraph entries and the corresponding comments are used, the paragraph names must begin within the A-Margin. The comment entries, on the other hand, must be wholly contained within the B-Margin. As with the IDENTIFICATION DIVISION, however, the comment entries begin immediately following the period that terminates the paragraph name. Unlike the entries in the IDENTIFICATION DIVISION, the entry itself is delimited by a period.

The SOURCE-COMPUTER Paragraph

The comment entry following this paragraph name identifies the computing system on which the program was developed, or the computer used to develop the source statements that constitute the program. Although the sole purpose of identifying the machine is documentation, the COBOL compiler does examine this entry to determine whether the computer specified is among its list of recognized computer systems.[1]

Statement Format

```
8
↓
SOURCE-COMPUTER. computername.
```

Various examples of this paragraph and its entry are shown in figure 15.1. In each case illustrated here, an IBM machine and model was specified. In all but the last example, such machines exist. The last example cites a machine not manufactured by IBM. Using the WATBOL COBOL compiler, this last entry would cause an error message.

Figure 15.1
Alternative SOURCE-COMPUTER entries

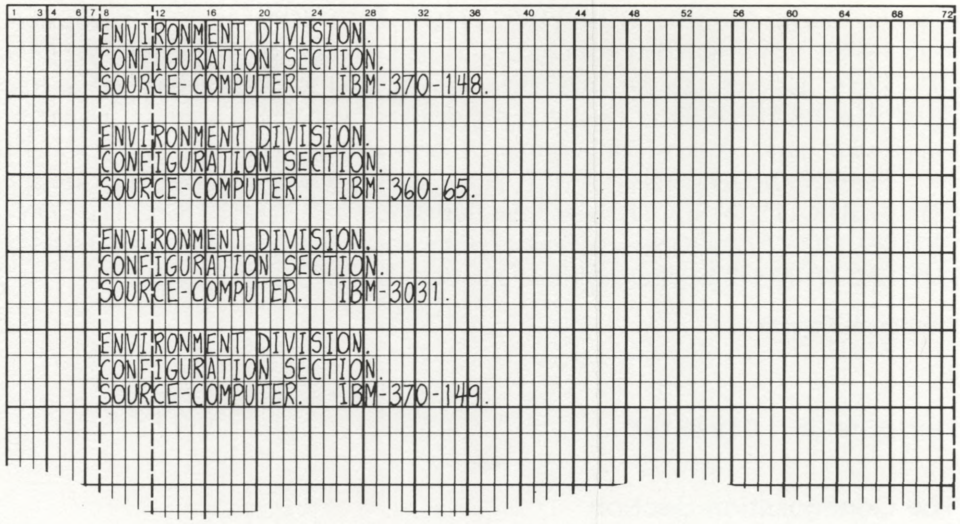
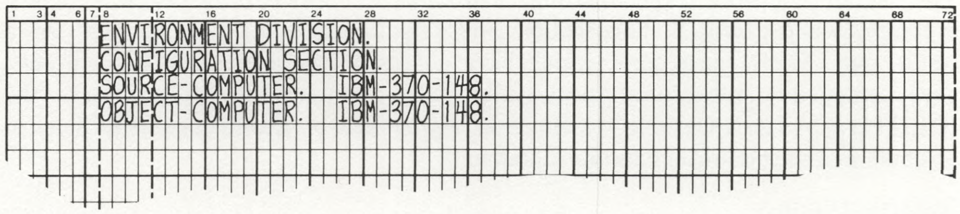


Figure 15.2
Typical SOURCE-COMPUTER and OBJECT-COMPUTER entries



The OBJECT-COMPUTER Paragraph

This paragraph entry describes the machine on which the COBOL program is to be executed following its development. Again, the COBOL compiler does not verify whether the machine exists or not. Typically, although not always, the computer specified as the source machine also serves as the object machine. Figure 15.2 illustrates both source and object machine entries.

Statement Format

```
8
↓
OBJECT-COMPUTER. computername.
```

The SPECIAL-NAMES Paragraph

This last paragraph in the CONFIGURATION SECTION serves a rather interesting set of purposes. One purpose is to establish programmer *mnemonic* names for special COBOL functions.

Mnemonic
A symbol that provides a COBOL name of the programmer's choosing for referring to a special function, like SYSIN or SYSOUT.

Statement Format

```
8
↓
SPECIAL-NAMES
    12
    ↓
    implementorname IS mnemonic....
```

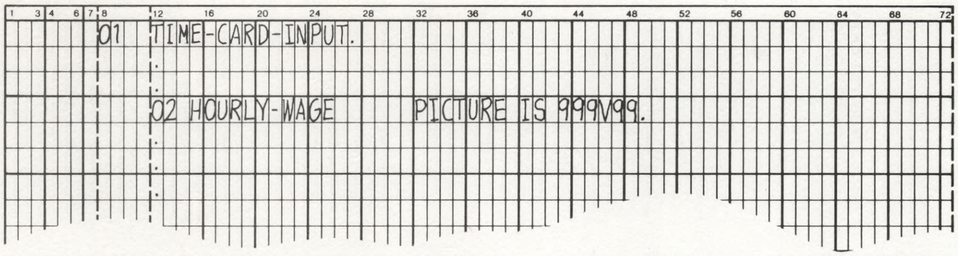
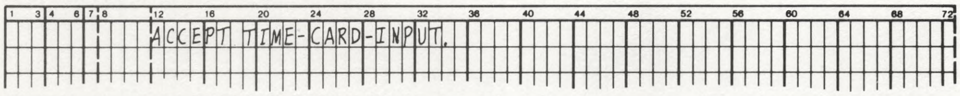


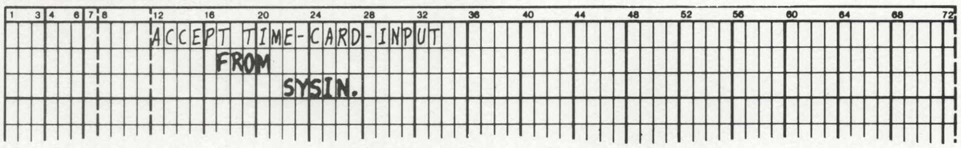
Figure 15.3
Record/variable descriptions as mnemonics

Figure 15.4
Alternatives of ACCEPT for reading record images

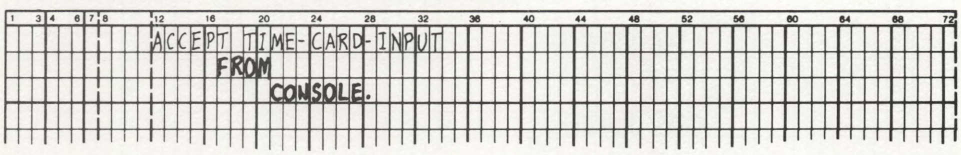
Alternative one: Abbreviated, default



Alternative two: Fully specified, confirms default



Alternative three: Fully specified, overrides default



A mnemonic is much like a variable name for a record or a field within a record. It is simply a tag, or programmer-assigned name, through which the associated item can be referred. Two examples of mnemonics can be seen in figure 15.3. The first, TIME-CARD-INPUT, allows the COBOL programmer to symbolically refer to the eighty character input record that contains time card information for an employee. The second, HOURLY-WAGE, performs the same function for a single field within the time card record.

COBOL special functions have, until now, been avoided. Again, they have been avoided because they are optional in some instances, and default values exist for them in other instances. The ACCEPT verb serves as an excellent example. The format of the ACCEPT verb, for reading card images, is shown in figure 15.4. Note that three alternatives are shown.

The FROM portion of the ACCEPT verb has been omitted thoughout this text. FROM describes the nature of the input device that will provide the COBOL program with the input record for processing. SYSIN is a function that specifies the input device will be a card reader. Thus, the form of the ACCEPT verb in alternative two will have time card records read from a card reader for processing. CONSOLE, used in alternative three of the ACCEPT verb, is a function whose name is a little more evident. This function specifies that the card images will be supplied by the computer operator's console.

The CONSOLE option is never used for processing regular input records. The computer operator has too many functions to perform to also feed executing programs their required data one card at a time. This leaves the SYSIN function as an alternative.

Figure 15.5
Use of SPECIAL-NAMES to declare mnemonics

```
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SOURCE-COMPUTER.  IBM-370.
       OBJECT-COMPUTER.  IBM-370.
       SPECIAL-NAMES.
           SYSIN IS CARD-READER.
```

Figure 15.6
Use of fuller forms of ACCEPT/DISPLAY verbs

```
       MAIN-CONTROL SECTION.
       M-C-001.
           ACCEPT TIME-CARD-INPUT
                FROM
                    CARD-READER.
           PERFORM PRINT-AND-READ-PROCESS
                UNTIL
                    TIME-CARD-INPUT = SPACES.
           STOP RUN.

       PRINT-AND-READ-PROCESS SECTION.
       P-A-R-P-001.
           MOVE CORRESPONDING TIME-CARD-INPUT TO
                                 TIME-CARD-OUTPUT.
           DISPLAY TIME-CARD-OUTPUT
                UPON
                    LINE-PRINTER.
           ACCEPT TIME-CARD-INPUT
                FROM
                    CARD-READER.
```

Figure 15.7
Use of SPECIAL-NAMES to declare mnemonic for DISPLAY

```
       SPECIAL-NAMES.
           SYSIN IS CARD-READER
           SYSOUT IS LINE-PRINTER.
```

The function SYSIN is unusually named. Very few programmers would elect SYSIN as their first choice in symbolic names for the computer system's card reader. Consequently, if the COBOL programmer wishes to rename this function, the SPECIAL-NAMES paragraph is used for just that purpose, as figure 15.5 illustrates. This function permits the use of ACCEPT verbs similar to the ones shown in figure 15.6. Figure 15.6 is a modified segment of the now familiar report program.

As you can see, the DISPLAY verb also has a more extensive specification than we have used thus far. Printed output can be displayed UPON a line printer (SYSOUT), the card punch (SYSPUNCH), or the operator's console (CONSOLE). Since we have dealt exclusively with line printers, and it is highly unlikely that we would choose to symbolically name the line printer SYSOUT, we need the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION. As a result, that segment of our COBOL program would actually look like figure 15.7. Note that the entries following the paragraph have only one period. Remember, the period terminates the statement entry. Consequently, only one period is used.

A word of caution regarding the choice of mnemonics for COBOL functions is in order. Remember, the programmer has been allowed flexibility in choosing symbolic names in COBOL programs. As with section, paragraph, and variable names, be careful to avoid names that can also be found on the list of reserved words established for the COBOL compiler you will be using.

Summary

The ENVIRONMENT DIVISION contains a CONFIGURATION SECTION whose paragraph and statement entries describe the compilation and operational environments of the COBOL program being developed. This section and its paragraphs are entirely optional, but like many such optional entries, they add valuable documentary information.

The section and paragraph names must be specified in the A-Margin, while the statements that correspond to each paragraph must reside in the B-Margin. B-Margin statements are terminated by periods. This is very different from the IDENTIFICATION DIVISION, where the use of periods in comment entries made little difference.

SOURCE-COMPUTER and OBJECT-COMPUTER paragraphs describe which computer system will be used in the COBOL program's development and execution, respectively. The SPECIAL-NAMES paragraph allows the programmer to choose more meaningful names for COBOL functions related to the ACCEPT and DISPLAY verbs.

Notes

1. Consult the technical reference materials for the COBOL compiler. These materials are available at your computing installation.

Questions for Review

1. What, if anything, is wrong with the following COBOL program segment?

```
ENVIRONMENT DIVISION.
OBJECT COMPUTER.  IBM-370-148.
SOURCE COMPUTER.  IBM-370.
SPECIAL-NAMES.
    SYSIN IS CARD-READER.
    SYSOUT IS LINE-PRINTER.
```

2. Reconstruct the report program described as an 80–80 listing in the summary of chapter 3 to include a complete IDENTIFICATION DIVISION and ENVIRONMENT DIVISION. Also, make sure that the fuller versions of the ACCEPT and DISPLAY verbs are used as well.

3. What advantage is gained by using a format of the ACCEPT verb that explicitly specifies the input device to be used to read input records? (Hint: See the title of Chapter 5.)

Problems

1. Examine the program shown in figure 3.14. Revise this program so that it possesses a fuller ENVIRONMENT DIVISION. In addition, restate all ACCEPT/DISPLAY instructions to use output devices whose names you may choose.

Programming Assignments

1. Revise the Senior Citizens edit report program from chapter 13 to include a fully specified ENVIRONMENT DIVISION.

2. Revise the Eglin Auto Parts edit report from chapter 13 to contain a fully specified ENVIRONMENT DIVISION.

16 Conventional File Structures

Chapter Outline

All input and output operations in the COBOL programs described in this textbook have been implemented using the ACCEPT and DISPLAY instructions, respectively. These two instructions are often termed *low volume I/O* statements.

Low Volume I/O
The ACCEPT and DISPLAY verbs in COBOL. They are used when few record images are to be read and printed, respectively. The programmer sacrifices control in detecting end of file and printer control for a smaller program.

Low volume I/O statements were created in COBOL so that programs requiring minimal input and/or output capabilities could carry out those functions without needing to be very large. The standard, or conventional, file processing capabilities in COBOL require the use of many, many statements in three of the four divisions of each program written. These statements cause even the smallest COBOL program to become quite large. A programmer desiring to read only several input records, for example, would be forced to write many COBOL statements if *conventional file structures* were used. The alternative, low volume I/O statements, requires only the use of the ACCEPT instruction to read input records and the DISPLAY instruction to print output records. ACCEPT and DISPLAY require special entries only in the PROCEDURE DIVISION.

Conventional File Structures
The READ and WRITE verbs in COBOL. When these verbs are used, COBOL programs are much larger. Extra statements are the price paid for greater control over the input and output devices used by the program.

The programs written thus far have made extensive use of low volume I/O statements. The reason for this is quite simple. Using low volume I/O statements requires the beginning student to learn far less COBOL than using conventional file instructions. Consequently, all input operations have been accomplished using the ACCEPT verb in COBOL. All printing operations have been accomplished using the DISPLAY verb.

Limitations of Low Volume Input/Output (I/O)

You might have the impression that the only difference between programs using low volume rather than conventional I/O is program size. The former yields smaller programs while the latter produces large programs. The differences, however, are much greater than that.

Printer Control

Using the DISPLAY instruction makes it nearly impossible to control the spacing operations of the printer on which our program outputs appear. Assume we wish to print a report title, followed by a blank line, followed by yet another report title. The sequence of COBOL instructions in figure 16.1 should look very familiar. The sequence is: (1) an entry in the WORKING-STORAGE SECTION named BLANK-LINE whose initial value is spaces and (2) a DISPLAY instruction to actually print the blank line. In fact, if four blank lines were required between the two titles, the DISPLAY instruction required to print the blank line would have to be repeated four times.

You may also have noticed that getting report titles to appear at the top of each physical page is almost impossible. Should you try to separate your output pages at the perforated seams you will not have a complete page of output. It is quite unlikely that the report titles will be at the top of each page and the report trailers at the bottom. This problem can be overcome. However, to do so requires very careful planning, line counting, and printing—not to mention a large number of DISPLAY instructions used solely to print blank lines.

```
           DISPLAY TITLE-1.
           DISPLAY BLANK-LINE.
           DISPLAY TITLE-2.
```

Figure 16.1
Use of DISPLAY verb for printing output

Detecting End of File

All of our programs are constructed with the assumption that the input records that are to be read and processed are followed by a blank record. In fact, the only way our program can detect the end of the input file is to encounter a blank record. While this is not as cumbersome or unwieldy as the problems with the DISPLAY verb, it is a problem nonetheless. In the real world, input files are not terminated by special records that have been added specifically to mark the end of the file. Why then do our programs make use of this device?

The ACCEPT verb is unable to detect when it has already read the last record in the input file. Every time the computer executes the ACCEPT verb the computer is instructed to assume a record does, in fact, exist and is ready to be read. If, on the other hand, no such record remains, the ACCEPT verb cannot successfully execute, and the program abnormally terminates. As a result, the use of the ACCEPT verb forces us to dedicate the very last record in an input file to be something other than just another record. This last record is assigned a value so unique that it is highly unlikely any other record in the file contains the same value. The value chosen is all blanks. Incorrect placement of the blank record in our input file would cause disastrous effects. Unfortunately, it is very easy to forget to supply the blank record or to misplace it in the file.

Defining Conventional Files

In order to overcome the limitations unique to low volume I/O statements just described, we shall describe how conventional files may be defined instead. This definition process requires us to reexamine two divisions in COBOL that have not been used for anything even resembling I/O operations. These are the ENVIRONMENT and DATA DIVISIONS.

A Modified Environment Division

In an earlier chapter we examined the ENVIRONMENT DIVISION and noted that this particular division also contains a CONFIGURATION SECTION. This division contains one other section, known as the INPUT-OUTPUT SECTION, as well. This section must contain an entry for each file a COBOL program is to process. Note the COBOL program segment in figure 16.2. The contents of the CONFIGURATION SECTION should be quite familiar, based on the content of chapter 14. This section is optional and need not appear in a COBOL program. However, it does describe the computing environment surrounding both the development and planned production usage of this particular program.

Figure 16.2
Function of the INPUT-OUTPUT SECTION

```
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SOURCE-COMPUTER.    IBM-370.
       OBJECT-COMPUTER.    IBM-370.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT TRANSACTION-FILE ASSIGN TO UR-2540-S-SYSIN.
           SELECT PRINT-FILE ASSIGN TO UR-1403-S-SYSPRINT.
```

Because INPUT-OUTPUT is a section, this entry must start within the A-Margin. Furthermore, because it is a section, the word SECTION must appear. One common error beginning programmers make is the ommission of the hyphen that joins the words INPUT and OUTPUT. Without the hyphen the COBOL compiler cannot recognize this section. You have already encountered this phenomenon in the WORKING-STORAGE SECTION. Unfortunately, some beginning programmers also adopt another notion, that of overhyphenating. This leads to the following entry, which is not correct: INPUT-OUTPUT-SECTION.

Statement Format

```
8
↓
INPUT-OUTPUT SECTION.
```

The INPUT-OUTPUT SECTION contains several paragraph entries. However, at this time only the FILE-CONTROL paragraph will be discussed. Here again, the programmer should be careful to supply the hyphen that joins the words FILE and CONTROL.

Statement Format

```
8
↓
FILE-CONTROL.
```

The FILE-CONTROL paragraph contains one entry for each file a program will operate. These file entries must begin and be wholly contained within the B-Margin. In addition, each entry must begin with the word SELECT. As the word implies, the programmer is selecting those files that the program is to process.

Statement Format

```
12
↓
SELECT filename ASSIGN TO implementorname.
```

In the PROCEDURE DIVISION, where files are actually manipulated, the files are referred to by their programmer-assigned names. These names are created using the same rules that govern the construction of section and paragraph names in the PROCEDURE DIVISION, and variable names. In the first SELECT clause the file chosen for processing is called TRANSACTION-FILE. This particular name was chosen because this file contains add and delete transactions that, following editing, will be used to modify a master file. The second SELECT clause was specified because our edit program must produce an edit report in which input record transactions, error messages, and other items of interest will appear.

The phrase ASSIGN TO follows the name of the file. While TRANSACTION-FILE and PRINT-FILE are *internal file names* to our program, each of these files can be found on some storage medium or physical I/O device in the computing system. The question is which one?

Internal File Name
The name immediately following SELECT. The name by which the file is referred throughout the PROCEDURE DIVISION.

These files may be found in many possible storage locations and I/O devices. The programmer informs the program, and hence the computing system, where such files are located. Note that the file our hypothetical edit program will read, known inside our COBOL program as TRANSACTION-FILE, can be found on/at UR-2540R-S-SYSIN. What does this mean?

The two characters, UR, designate the file is to reside on what is known as a *Unit Record device.* Unit record devices include card readers, card punches, and line printers. Two other designations include DA for Direct-Access storage devices, and UT for UTility devices, such as a magnetic tape.

Unit Record Device
An I/O device that reads or writes unit records. Examples of these devices are card readers, card punches, and line printers.

The next part of the code that describes the physical I/O device names the model number of that device. A 2540R is one model of card reader. A 1403 is one model of line printer. Your computing installation may use other models and you should, therefore, consult your instructor for the proper choice of model numbers to use.

The letter *S* in both SELECT clauses indicates the device is to be used in a sequential mode of operation. For card readers this means the first card will be read followed by the second, and then the third, fourth, and so on. The same is true of line printers. The first line of output will be followed by the second, etc. In fact, it is hard to imagine using unit record devices in any other manner. This is not true, however, for all devices, such as the direct-access storage device. Consequently, the letter *S* is not the only letter possible.

The final portion of the description is the name of the job control statement that provides other identifying characteristics of the file or the device that hosts the file. In our examples shown earlier, the source deck containing the COBOL source statements for the edit program also contains several job control cards that further describe the TRANSACTION-FILE and the PRINT-FILE. Those job control cards have been assigned the symbolic names of SYSIN and SYSPRINT, respectively.

The first SELECT clause names the input file TRANSACTION-FILE, which is the name used for that file in the PROCEDURE DIVISION. This file is located on a unit record device whose model number is a 2540R card reader. The device is to be operated in a sequential manner. Other information regarding this file/device can be found on the accompanying job control statement whose symbolic name is SYSIN. A similar interpretation can be constructed for the second SELECT clause that defines the PRINT-FILE.

A Modified Data Division

Buffer
A staging, or holding, area in primary memory where record images are held before printing or immediately after reading.

Conventional file structures also require that buffer space be allocated for each file mentioned in the INPUT-OUTPUT SECTION. *Buffers* are areas of computer memory where record images are held after having been read by the card reader. For print files, buffers hold print-line images prior to being printed. The low volume I/O statements, ACCEPT and DISPLAY, provided buffer space automatically.

Buffer areas are reserved in a section of the DATA DIVISION known as the FILE SECTION. A FILE SECTION appropriate to the developing example is shown in figure 16.3. The FILE SECTION entry must, like all other sections, start in the A-Margin. A common error here is to add a hyphen where none is needed, yielding FILE-SECTION, which is incorrect.

Statement Format

```
8
↓
FILE SECTION.
```

Each file selected in the INPUT-OUTPUT SECTION must have a file definition in the FILE SECTION. Each file definition (FD) starts in the A-Margin. The name of the file starts in the B-Margin. Notice that the name is not followed immediately by a period. Instead, the phrase LABEL RECORDS ARE OMITTED follows, wholly contained in the B-Margin, and then the period occurs.

```
       DATA DIVISION.
           SKIP2
       FILE SECTION.
           SKIP1
       FD  TRANSACTION-FILE
           LABEL RECORDS ARE OMITTED.
       01  TRANSACTION-RECORD              PICTURE IS X(80).
           SKIP1
       FD  PRINT-FILE
           LABEL RECORDS ARE OMITTED.
       01  PRINT-RECORD                    PICTURE IS X(120).
```

FIGURE 16.3
Addition of the FILE SECTION

Statement Format

```
8   12
↓   ↓
FD  filename
                       {STANDARD}
    LABEL RECORDS ARE  {OMITTED } .
01  recordname  PICTURE IS attributes.
```

Certain physical I/O devices can hold more than one file simultaneously. Magnetic tape is an excellent example. Just as the tape in home tape recorders can hold many songs, each tape can hold more than one file. Accordingly, each file is prefaced by what is known as a label record. Label records list the name of the file for identification purposes as well as other useful items of information. Direct-access storage devices also make use of label records for the very same reason. Line printers and card readers, however, can only hold one file at a time. This is the reason for the phrase LABEL RECORDS ARE OMITTED. Since these prefatory records are not present, the computer system need not set aside buffer space that would be used only to hold these label records.

To complete each file definition, the programmer must supply a model record description. This prototype describes a very important characteristic of each record in the file: its length. In the case of the TRANSACTION-FILE each record is 80 characters long; in the PRINT-FILE each record is 120 characters long. These record descriptions are like all record descriptions. The 01 starts in the A-Margin. A unique name must be supplied, and the PICTURE IS clause describes the record's attributes.

The current version of our edit program now using conventional file structures is partially shown in figure 16.4. The divisions shown include the IDENTIFICATION (unchanged), ENVIRONMENT (changed), and the DATA DIVISION (changed). This last division is only partially shown. Since the WORKING-STORAGE SECTION remains unaffected by the use of conventional file structures, it is not repeated in its entirety. Note that using conventional I/O statements has caused this portion of our COBOL program to grow by eleven statements. The increase, fortunately, will prove more than bearable when the added flexibility of conventional I/O instructions becomes evident.

The next set of significant program alterations must occur in the PROCEDURE DIVISION of our COBOL program.

Figure 16.4
Complete IDENTIFICATION, ENVIRONMENT, and DATA DIVISIONS

Page 1 of 3

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.  EDT1601.
       AUTHOR.     HOWARD D. WEINER
                       A PROFESSOR OF COMPUTER SCIENCE
                       DEPARTMENT OF MATHEMATICS, STATISTICS &
                          COMPUTER SCIENCE
                       COLLEGE OF ARTS AND SCIENCES
                       THE AMERICAN UNIVERSITY
                       WASHINGTON, DC 20016
       INSTALLATION.  ACADEMIC COMPUTER SERVICES
                      DIVISION OF COMPUTER SERVICES
                      THE AMERICAN UNIVERSITY
                      WASHINGTON, DC 20016
       DATE-WRITTEN.  12 APRIL 1980
       DATE-COMPILED.
       SECURITY.   NONE.
       REMARKS.    THIS IS AN EDIT PROGRAM THAT IS DESIGNED TO
                   VERIFY THE ACCURACY OF ADD AND DELETE
                   TRANSACTIONS THAT WILL SUBSEQUENTLY BE
                   APPLIED TO A FURNITURE MASTER FILE IN AN
                   INVENTORY APPLICATION.
           SKIP1
                   THE PROGRAM IS CONSTRUCTED FROM A PROGRAM
                   LOGIC MODEL FOR EDIT PROGRAMS DISCUSSED
```

Page 2 of 3

```
                   IN CHAPTER 12 THE PROGRAM WAS ORIGINALLY
                   DEVELOPED IN AN EARLIER CASE STUDY.
           SKIP3
           SKIP3
       ENVIRONMENT DIVISION.
           SKIP2
       CONFIGURATION SECTION.
       SOURCE-COMPUTER.   IBM-370.
       OBJECT-COMPUTER.   IBM-370.
           SKIP1
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT TRANSACTION-FILE ASSIGN TO UR-2540R-S-SYSIN.
           SELECT PRINT-FILE ASSIGN TO UR-1403-S-SYSPRINT.
           SKIP3
           SKIP3
       DATA DIVISION.
           SKIP2
       FILE SECTION.
           SKIP1
       FD  TRANSACTION-FILE
           LABEL RECORDS ARE OMITTED.
       01  TRANSACTION-RECORD       PICTURE IS X(80).
           SKIP1
```

Page 3 of 3

```
       FD  PRINT-FILE
           LABEL RECORDS ARE OMITTED.
       01  PRINT-RECORD             PICTURE IS X(120).
           SKIP3
       WORKING-STORAGE SECTION
      *
      *     PROGRAM COUNTERS AND STATE VARIABLES
      *
       01  TRANSACTION-STATUS       PICTURE IS 9.
           88 TRANSACTION-IS-OK     VALUE 0.
           88 TRANSACTION-IS-BAD    VALUE 1.
```

```
       OPEN INPUT TRANSACTION-FILE.
       OPEN OUTPUT PRINT-FILE.
```

or

```
       OPEN INPUT TRANSACTION-FILE
            OUTPUT PRINT-FILE.
```

Figure 16.5
Format of OPEN verb

Manipulating Conventional Files

The PROCEDURE DIVISION is the division where files are actually manipulated during the execution of a COBOL program. Our files were only defined in the two earlier divisions. To operate upon these files requires the introduction of four new COBOL verbs.

Readying Files for Processing

The ACCEPT and DISPLAY verbs can be executed as soon in the COBOL program as the programmer desires. No preparation is required, because these low volume I/O statements automatically acquire buffer space. Unfortunately, conventional file processing instructions require formal preparation that must explicitly be requested by the COBOL program before a file is read from or written to. These preparations are initiated by use of the OPEN verb, whose formats are shown in figure 16.5. The OPEN verb causes buffer space to be allocated for the file or files named in the instruction.

Statement Format

```
12
↓
      {INPUT }
OPEN  {OUTPUT}  filename.
```

Just how much space is required and for what purpose can be learned from the FILE SECTION in the DATA DIVISION. Here each FD contains a prototypical record whose length determines the size of each buffer acquired when the file is opened. Further, had LABEL RECORDS ARE STANDARD (rather than OMITTED) been specified, a buffer to contain the label record in the file would have been acquired as well.

Also note that immediately following the verb the programmer specifies whether the file is to be processed as an input, or output file. Files are opened as either INPUT or OUTPUT. A file opened as INPUT, such as TRANSACTION-FILE, will have its records read. OUTPUT files, such as PRINT-FILE, will have records, or lines of print, written to them.

The OPEN verb, like all other verbs, must be wholly contained within the B-Margin. The programmer can open one or more files with each OPEN instruction. Any attempt to process a file before it has been opened will result in an abnormal termination. A file cannot be read if buffer space for that file has not been acquired.

Figure 16.6
Format of CLOSE verb

```
           CLOSE TRANSACTION-FILE.
           CLOSE PRINT-FILE.
```

or

```
           CLOSE TRANSACTION-FILE
                 PRINT-FILE.
```

Terminating File Processing

At some time in each program's execution, it no longer needs to access one or more of its files. When a file is no longer needed, the programmer must take steps to relinquish access and thereby give up the file's buffers. To accomplish this task, COBOL programs must use the CLOSE verb as shown in figure 16.6. The CLOSE verb is followed immediately by the name of the file to be closed. Closing a file does not require specifying whether the file is an input file or an output file. That information is available from the OPEN verb. As with the OPEN verb, one or more files may be CLOSED with a single instruction.

Statement Format

```
12
↓
CLOSE filename.
```

When a file is closed, it relinquishes its buffer space. If a file has no buffers assigned to it, any attempt to read from it or write to it will result in abnormal termination of the program. Thus, all access requests must occur after the file has been opened, but before it has been closed.

Replacing the ACCEPT Verb with READ

Rather than ACCEPT, input records in our COBOL programs, we will now READ the input file. The format of this particular verb is shown in figure 16.7. This instruction causes one input record from a file, known within our COBOL program as TRANSACTION-FILE, to be placed in a record description in the WORKING-STORAGE SECTION named INVENTORY-RECORD-INPUT.

The ACCEPT verb shown in figure 16.8 causes the same result. What is the difference between the two instructions? After all, if the low volume I/O instruction causes the same action to occur, why use the conventional I/O instruction and its accompanying entries in the ENVIRONMENT, DATA, and PROCEDURE DIVISIONS?

In all our earlier examples, our input files contained a blank record as the last record. Like its nonblank predecessors, this record was read by the ACCEPT verb. This blank record was read during the very last part of the edit or report process in our edit or report programs, respectively. Thus, the statement in the MAIN-CONTROL SECTION shown in figure 16.9 looked for a blank record. If a blank record was found, the program completed its execution; if the last record read was not blank, execution continued.

```
READ TRANSACTION-FILE
    INTO
        INVENTORY-RECORD-INPUT
    AT END
        MOVE SPACES TO INVENTORY-RECORD-INPUT.
```

Figure 16.7
Format of the READ instruction

```
ACCEPT INVENTORY-RECORD-INPUT.
```

Figure 16.8
More familiar ACCEPT verb

```
PERFORM EDIT-PROCESS
    UNTIL
        INVENTORY-RECORD-INPUT = SPACES.
```

Figure 16.9
Iterative instruction in search of a blank record

By using the conventional I/O instruction shown in figure 16.7 the last record in the file will not be blank. In this instance, blank records are no longer needed to mark the end of a file. Spaces will be moved to INVENTORY-RECORD-INPUT when the READ instruction attempts to read another record and all of the records in the file have already been read. Now the file will be AT END. AT END spaces are moved to INVENTORY-RECORD-INPUT, which makes it appear as though a blank record was, in fact, read.

Statement Format

```
12
↓
READ filename [INTO recordname]
     AT END statement(s).
```

The ACCEPT instruction, on the other hand, abnormally terminates if we attempt to read another record and no more input records remain to be read. This is why the blank record is placed at the very end of every file processed by an ACCEPT instruction. The READ instruction, however, can be executed if no more input records remain in the input file.

The AT END clause of the READ instruction will only receive control when the computer attempts to read yet another input record and no more records are available to be read. As the earlier example indicated, when the end of file has been found, the READ instruction will cause spaces to be moved into the WORKING-STORAGE variable INVENTORY-RECORD-INPUT. Consequently, the iterative instruction in the main control portion of our report and edit programs remains unchanged.

Replacing the DISPLAY Verb with WRITE

The remaining low volume I/O instruction to replace is the DISPLAY instruction. Rather than displaying reformatted input records, titles, trailers, error messages, and summary totals, we shall now WRITE them.

COBOL has a rather strange convention that is not always clear to students of the language. Specifically, while we read files, we write records. This means that the conventional I/O instruction to print a report title appears as shown in figure 16.10. It might seem more natural to state WRITE PRINT-FILE. . . . After all, we did READ TRANSACTION-FILE. However, one writes records, not files, in COBOL.

The WRITE instruction transfers the data in the WORKING-STORAGE variable named TITLE-2 to the buffer allocated to PRINT-FILE. That buffer is named PRINT-RECORD. The resulting line of print is not actually printed until the printer has first advanced two lines. The actual line of print appears on the second line. Hence, stating AFTER ADVANCING 2 LINES provides double-spaced print. If we desire single-spaced output instead, we could specify AFTER ADVANCING 1 LINES. Note that the word LINES is always plural even when only one line is being advanced. If we desire triple-spaced output instead, we could specify AFTER ADVANCING 3 LINES.

Statement Format

```
12
↓
WRITE filerecordname [FROM recordname]
                        ⎧PAGE     ⎫
     AFTER ADVANCING    ⎨mnemonic ⎬ .
                        ⎩n LINES  ⎭
```

The ability to skip lines on the line printer is a very powerful tool. Each line skipped over is a blank line. This means we can revise our earlier COBOL programs to delete all DISPLAY statements whose function it was to print blank lines. In fact, we can remove the declaration of the variable BLANK-LINE from the WORKING-STORAGE SECTION.

At the beginning of this chapter, in figure 16.1, a hypothetical series of COBOL instructions were grouped together to print two titles, each separated by a blank line. This required three DISPLAY instructions. This can be rewritten using conventional file instructions as illustrated in figure 16.11. The second WRITE instruction not only causes the second title to be printed, but it also causes the intervening blank line to be produced as well.

Figure 16.10
Format of the WRITE verb

```
           WRITE PRINT-RECORD
               FROM
                   TITLE-2
               AFTER ADVANCING
                   2 LINES.
```

Figure 16.11
WRITE statements to print report titles

```
           WRITE PRINT-RECORD
               FROM
                   TITLE-1
               AFTER ADVANCING
                   1 LINES.
           WRITE PRINT-RECORD
               FROM
                   TITLE-2
               AFTER ADVANCING
                   2 LINES.
```

Special Printer Control

It is also possible to use conventional I/O instructions to print report titles at the very top of the physical page and trailers at the very bottom. The DISPLAY instruction made this difficult, if not impossible.

COBOL features special functions known as *printer channels.* Twelve channels can be defined at every computing installation. A channel is a definition of a particular line on the printed page.

Printer Channels
The hardware device within the printer that allows the computing installation to control the processing of paper stock.

Assume a printed page presents sixty-five possible horizontal print lines. A computing installation could define the topmost line on each page as a channel. This would most likely be channel 1. Since only twelve such channels are possible, the very last horizontal line on the page could be defined as channel 12. The remaining ten channels could be defined as other lines between the topmost and bottommost lines.

Both the very top and very bottom lines on a printed page are close to the perforations that permit continuous paper forms to be separated into single, detached pages. The tearing process might disfigure the very top and bottom lines on each page. As a result, line 3 might be considered the first line on the page on which anything should be printed. Correspondingly, line 60 might be defined as the bottom line for printing purposes. In this case, channel 1 would be line 3 and channel 12 would be line 60.

Special functions, such as channel numbers, must be given symbolic names when used in COBOL programs. All symbolic names for special functions are defined in the SPECIAL-NAMES paragraph of the CONFIGURATION SECTION in the ENVIRONMENT DIVISION. To make use of greater printer control, our COBOL programs might be modified to look similar to the program shown in figure 16.12.

If we desire to print report titles only at the top of each page, the COBOL WRITE instructions in figure 16.13 might be used.

```
       ENVIRONMENT DIVISION.
           SKIP2
       CONFIGURATION SECTION.
           SKIP1
       SOURCE-COMPUTER.  IBM-370.
       OBJECT-COMPUTER.  IBM-370.
       SPECIAL-NAMES.
           C01 IS TOP-OF-PAGE
           C12 IS BOTTOM-OF-PAGE.
           SKIP2
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT TRANSACTION-FILE ASSIGN TO UR-2540R-S-SYSIN.
           SELECT PRINT-FILE ASSIGN TO UR-1403-S-SYSPRINT.
```

Figure 16.12
SPECIAL-NAMES entries for printer control

```
       WRITE PRINT-RECORD
           FROM
               TITLE-1
           AFTER ADVANCING
               TO-TOP-OF-PAGE.
       WRITE PRINT-RECORD
           FROM
               TITLE-2
           AFTER ADVANCING
               2 LINES.
```

Figure 16.13
Using special printer control in WRITE statement

Figure 16.14
Using special printer control to print trailers

```
           ADD 1 TO PAGE-NUMBER OF PROGRAM-VARIABLES.
           MOVE PAGE-NUMBER OF PROGRAM-VARIABLES TO
                   PAGE-NUMBER OF TRAILER-3.
           WRITE PRINT-RECORD
               FROM
                   TRAILER-3
               AFTER ADVANCING
                   TO-BOTTOM-OF-PAGE.
```

Since page numbers are frequently the very last items to appear on the printed page, we might want them printed by using the series of instructions shown in figure 16.14. You are advised to determine the channel definitions in effect at your computing installation prior to making use of them.

A reserved word in COBOL, PAGE, is a built-in definition of channel 1 for the printer. Unlike the function, CO1, however, our WRITE verb can use PAGE in the WRITE verb. The SPECIAL-NAMES paragraph is not needed when PAGE is used. Indeed, all that is required is the phrase AFTER ADVANCING PAGE.

Processing in Buffers

Both the READ and WRITE instructions cause record images to be transferred INTO or FROM the WORKING-STORAGE SECTION. In our report and edit programs, the variables manipulated reside in the WORKING-STORAGE SECTION. Buffers are used only as a stopover between WORKING-STORAGE and the physical I/O device. It is possible, however, to process records directly in the buffers, thus eliminating the need for certain WORKING-STORAGE variables.

Buffer Processing for Input Files

If we desire to edit input transactions in the buffers into which they have been read, we remove the record definition INVENTORY-RECORD-INPUT from the WORKING-STORAGE SECTION and place it in the file definition (FD) for TRANSACTION-FILE. This yields a FILE SECTION that appears in figure 16.15.

The real gain from relocating the record description in this manner is in all of the READ instructions that accessed TRANSACTION-FILE. Each READ instruction can be performed in fewer lines of COBOL coding as figure 16.16 illustrates. What is missing? The INTO clause is not needed because the record image read is not being transferred into the WORKING-STORAGE SECTION. Instead it is being processed directly in the buffer. In the past, input operations always caused a record image first to be placed into a buffer and then into a record description in the WORKING-STORAGE SECTION. By placing the record description as part of the file definition (FD), the second step is eliminated.

This causes one slight problem, however. Note that the AT END clause attempts to move spaces into the buffer (INVENTORY-RECORD-INPUT is now the buffer record) if the READ was unsuccessful because the end of the file has been reached. A buffer is not available for such use if the end of file has occurred. Buffers can only be accessed if the read instruction yields a real record from the input file. This problem can be easily overcome.

```
DATA DIVISION
    SKIP2
FILE SECTION
    SKIP1
FD  TRANSACTION-FILE
    LABEL RECORDS ARE OMITTED.
01  INVENTORY-RECORD-INPUT.
    02 INVENTORY-NUMBER     PICTURE IS X(06).
    02 TRANSACTION-CODE     PICTURE IS X(01).
       88 ADD-CODE          VALUE "A".
       88 DELETE-CODE       VALUE "D".
    02 QUANTITY             PICTURE IS X(03).
    02 DESCRIPTION          PICTURE IS X(25).
    02 FILLER               PICTURE IS X(45).
    SKIP1
FD  PRINT-FILE
    LABEL RECORDS ARE OMITTED.
01  PRINT-RECORD            PICTURE IS X(120).
```

Figure 16.15
Record description moved to buffer area

```
    READ TRANSACTION-FILE
        AT END
            MOVE SPACES TO INVENTORY-RECORD-INPUT.
```

Figure 16.16
Reading record images directly into a file buffer

```
01  INPUT-FILE-STATUS       PICTURE IS X(03) VALUE "NO".
    88 FILE-IS-EMPTY        VALUE "YES".

    PERFORM EDIT-PROCESS
        UNTIL
            FILE-IS-EMPTY.

    READ TRANSACTION-FILE
        AT END
            MOVE "YES" TO INPUT-FILE-STATUS.
```

Figure 16.17
Detecting end of file with READ verb

Should you desire to process input file records directly in their buffers rather than in WORKING-STORAGE, you must define a bi-stable variable that notes whether the end of the file has been encountered or not. (See figure 16.17.) The segments in figure 16.17 illustrate the use of a character-implemented bi-stable device to detect when the end of file has occurred. The AT END clause in the READ instruction is activated only when a READ has been attempted and no records remain to be read. Then the value YES will replace NO in INPUT-FILE-STATUS. For all earlier READS the AT END clause will remain inactive and INPUT-FILE-STATUS will possess the value NO. The iterative instruction that controls the entire editing process continues until an end of file condition has been reached. When INPUT-FILE-STATUS contains the value YES, the condition FILE-IS-EMPTY is said to exist. When this condition occurs, the edit process is ended.

Processing Buffers for Output Files

The only output files examined so far have been print files. Print files pose a special problem if we attempt to process records directly in buffers rather than in the WORKING-STORAGE SECTION. The nature of the problem relates to the VALUE clause found in variable definitions.

Buffers are not acquired for any file until the file is activated by an OPEN verb. Record descriptions, however, are defined and given initial values during translation of the source program by the COBOL compiler. Consequently, the VALUE clause in a record description in a buffer area cannot be accommodated. How can the variable be given the initial value when it is part of a buffer that will not exist until the containing file is opened? Files are opened during execution. VALUE clauses are operated upon during compilation by the COBOL compiler.

Many of the record descriptions associated with our print files are report titles and trailers. Both titles and trailers contain many VALUE clauses. An attempt to define a title or trailer in a file buffer area will result in a compilation error message issued by the COBOL compiler and the VALUE clause being ignored. For this reason we shall continue to place our record descriptions for print files in the WORKING-STORAGE SECTION of our COBOL programs rather than in the buffer definitions that are part of the FILE SECTION.

Summary

Conventional file processing instructions provide greater, more realistic control over file processing than do the more familiar low-volume I/O instructions. Using the READ instruction, as opposed to ACCEPT, permits us to discontinue placing blank records at the end of files to be read by our COBOL programs. We no longer need to fear that our programs will abnormally terminate or end prematurely by misplacing or forgetting the blank record. The WRITE instruction offers greater printer control than its low-volume counterpart, the DISPLAY instruction. By using WRITE, our programs can skip lines and advance to the top or bottom of the printed page without having to generate intervening blank lines.

Unfortunately, conventional file processing capabilities require the inclusion of many more statements in our COBOL programs. Major additions to the ENVIRONMENT, DATA, and PROCEDURE DIVISIONS must be made.

The ENVIRONMENT DIVISION must be expanded to feature a new section. This section is known as the INPUT-OUTPUT SECTION, and it is composed of a FILE-CONTROL paragraph. The FILE-CONTROL paragraph contains one SELECT statement for each file a COBOL program is to process. Each file to be used must be specified by a SELECT statement. SELECT statements provide the name by which the file will be referred to in the program. It also provides descriptive characteristics of the device upon which the file is located.

The DATA DIVISION also contains a new section, the FILE SECTION. The FILE SECTION contains a file definition (FD) for each file specified in a SELECT statement in the INPUT-OUTPUT SECTION of the ENVIRONMENT DIVISION. Each FD is followed by the file's name and a specification as to whether file label records exist for the file or not. Typically, unit record files do not possess label records. Files stored on magnetic tape or direct-access storage devices, on the other hand, must possess label records because they can hold more than one file at a time. When multiple files are present on a storage medium, those files should be prefaced by a label record that identifies each file. Finally, the FD is completed by a prototypical record definition that describes the characteristics (i.e., length) of each record in the file.

The PROCEDURE DIVISION must be examined for all ACCEPT and DISPLAY instructions. These must be replaced by READ and WRITE instructions, respectively. The WRITE instruction permits us to remove all the DISPLAY instructions for printing blank lines. Blank lines can now be generated automatically by specifying how many blank lines are desired in the AFTER ADVANCING clause of the WRITE instruction.

Every computing installation features a printer that can have channel controls defined. These channels, when specified in a COBOL program, cause the printer to advance to a particular part of the paper stock prior to printing. A maximum of twelve such channels may be defined. Typically, channel 1 is defined to be at the topmost position of the page. Channel 12 is usually the bottommost part of each page.

All channels you desire to use in a COBOL program must be assigned symbolic names. These names are assigned in the SPECIAL-NAMES paragraph of the CONFIGURATION SECTION of the ENVIRONMENT DIVISION.

Programmers have the option of processing certain records in the WORKING-STORAGE SECTION of a COBOL program or in the buffer areas that are a part of each FD in the FILE SECTION. If records are processed in the buffer areas, the appropriate record descriptions should be relocated from the WORKING-STORAGE SECTION to the FILE SECTION. However, record descriptions containing variables assigned initial values through the VALUE clause cannot be a part of the FILE SECTION. This eliminates processing report titles and trailers directly in the buffers reserved for print files.

The restriction on the use of the VALUE clause stems from the time at which buffers are actually acquired for files. Files are allocated buffers when they are opened, using the COBOL OPEN verb. The VALUE clause is only operational during compilation, and compilation precedes execution. This would require the VALUE clause to be made operational during execution rather than compilation, which is impossible.

Any reference to a variable that is part of a record description in a file buffer area can only occur if buffers have been allocated. This restricts variable access to the time after a file is opened using the OPEN verb. Furthermore, once the file is closed by the CLOSE verb, the buffers that had been acquired are then relinquished. Consequently, variables in record descriptions that are themselves located in file buffer areas cannot be accessed after the file is closed.

The edit program developed earlier appears in figure 16.18. It has been modified to use conventional rather than low volume I/O instructions.

Notes

1. There are two ways to include job control statements along with COBOL source programs. The first method involves keypunching each job control statement and placing it in the proper position within your job deck. This is quite cumbersome since the COBOL compiler requires many such statements. The second method involves the use of catalogued procedures.

   Catalogued procedures are sets of commonly used job control statements. Sometimes called *procs,* these statements can automatically be made a part of your job deck by specifying the name under which they have been stored in the catalogue procedure library.

   If the first method is used, then you should be able to match SELECT statements with job control statements—both are part of the job deck. The second method is likely to hide many of the job control statements from you, since they are stored in the catalogue procedure library.

Figure 16.18
Edit program using conventional file structures

Page 1 of 18

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. EDT1602.
       AUTHOR.    HOWARD D. WEINER
                  A PROFESSOR OF COMPUTER SCIENCE
                  DEPARTMENT OF MATHEMATICS, STATISTICS &
                      COMPUTER SCIENCE
                  COLLEGE OF ARTS AND SCIENCES
                  THE AMERICAN UNIVERSITY
                  WASHINGTON, DC 20016
       INSTALLATION.    ACADEMIC COMPUTER SERVICES
                        DIVISION OF COMPUTER SERVICES
                        THE AMERICAN UNIVERSITY
                        WASHINGTON, DC 20016
       DATE-WRITTEN.    12 APRIL 1980
       DATE-COMPILED.
       SECURITY.  NONE.
       REMARKS.     THIS IS AN EDIT PROGRAM THAT IS DESIGNED TO
                    VERIFY THE ACCURACY OF ADD AND DELETE
                    TRANSACTIONS THAT WILL SUBSEQUENTLY BE
                    APPLIED TO A FURNITURE MASTER FILE IN AN
                    INVENTORY APPLICATION.
           SKIP1
                    THE PROGRAM IS CONSTRUCTED FROM A PROGRAM
                    LOGIC MODEL FOR EDIT PROGRAMS DISCUSSED
```

Page 2 of 18

```
                    IN CHAPTER 12.
           SKIP3
           SKIP3
       ENVIRONMENT DIVISION.
           SKIP2
       CONFIGURATION SECTION.
       SOURCE-COMPUTER.   IBM-370.
       OBJECT-COMPUTER.   IBM-370.
       SPECIAL-NAMES.
           C01 IS TO-TOP-OF-PAGE,
           C12 IS TO-BOTTOM-OF-PAGE.
           SKIP1
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT TRANSACTION-FILE ASSIGN TO UR-2540R-S-SYSIN.
           SELECT PRINT-FILE       ASSIGN TO UR-1403-S-SYSPRINT.
           SKIP3
           SKIP3
       DATA DIVISION.
           SKIP2
       FILE SECTION.
           SKIP1
       FD  TRANSACTION-FILE
```

```
           LABEL RECORDS ARE OMITTED.
       01  TRANSACTION-RECORD      PICTURE IS X(80).
           SKIP1
       FD  PRINT-FILE
           LABEL RECORDS ARE OMITTED.
       01  PRINT-RECORD            PICTURE IS X(120).
           SKIP3
       WORKING-STORAGE SECTION.
      *
      *     PROGRAM COUNTERS AND STATE VARIABLES
      *
       01  TRANSACTION-STATUS      PICTURE IS 9 VALUE 0.
           88 TRANSACTION-IS-OK    VALUE 0.
           88 TRANSACTION-IS-BAD   VALUE 1.
           SKIP1
       01  SUBTOTALS.
           02 ERRORS-FOUND         PICTURE IS 9(04) VALUE 0.
           02 TRANSACTIONS-WITH-ERRORS
                                   PICTURE IS 9(04) VALUE 0.
           02 TRANSACTIONS-READ    PICTURE IS 9(04) VALUE 0.
           02 CURRENT-LINE         PICTURE IS 9(02) VALUE 0.
              88 PAGE-OVERFLOW     VALUE 41 THRU 99.
           02 PAGE-NUMBER          PICTURE IS 9(04) VALUE 0.
           SKIP2
```



```
      *
      *     REPORT TITLES AND TRAILERS
      *
       01  TITLE-1.
           02 FILLER                       PICTURE IS X(53) VALUE SPACES.
           02 FILLER                       PICTURE IS X(67) VALUE
           "COX INDUSTRIES".
           SKIP1
       01  TITLE-2.
           02 FILLER                       PICTURE IS X(52) VALUE SPACES.
           02 FILLER                       PICTURE IS X(68) VALUE
           "INVENTORY SYSTEM".
           SKIP1
       01  TITLE-3.
           02 FILLER                       PICTURE IS X(54) VALUE SPACES.
           02 FILLER                       PICTURE IS X(66) VALUE
           "EDIT  REPORT".
           SKIP1
       01  TITLE-4.
           02 FILLER                       PICTURE IS X(20) VALUE SPACES.
           02 FILLER                       PICTURE IS X(40) VALUE
           ".........1.........2.........3.........4".
           02 FILLER                       PICTURE IS X(60) VALUE
           ".........5.........6.........7.........8".
```

Figure 16.18
(continued)


```
           SKIP1
       01  TRAILER-1.
           02 FILLER                PICTURE IS X(20) VALUE SPACES.
           02 FILLER                PICTURE IS X(40) VALUE
           "..........1.........2.........3.........4".
           02 FILLER                PICTURE IS X(60) VALUE
           "..........5.........6.........7.........8".
           SKIP1
       01  TRAILER-2.
           02 FILLER                PICTURE IS X(55) VALUE SPACES.
           02 FILLER                PICTURE IS X(05) VALUE
           "PAGE-".
           02 PAGE-NUMBER           PICTURE IS Z,ZZ9.
           02 FILLER                PICTURE IS X(55) VALUE SPACES.
           SKIP2
      *
      *     INPUT RECORD AND OUTPUT RECORD DEFINITIONS.
      *
       01  INVENTORY-RECORD-INPUT.
           02 INVENTORY-NUMBER      PICTURE IS X(06).
           02 TRANSACTION-CODE      PICTURE IS X(01).
               88 VALID-TRANSACTION-CODE
                                    VALUE "A", "D".
           02 QUANTITY              PICTURE IS X(03).
```



```
           02 DESCRIPTION           PICTURE IS X(25).
           02 FILLER                PICTURE IS X(45).
           SKIP1
       01  INVENTORY-RECORD-OUTPUT.
           02 FILLER                PICTURE IS X(20) VALUE
           " RECORD READ -->".
           02 RECORD-IMAGE          PICTURE IS X(100).
           SKIP2
      *
      *     SUMMARY STATISTICS RECORD DEFINITIONS.
      *
       01  SUMMARY-1.
           02 FILLER                PICTURE IS X(20) VALUE SPACES.
           02 FILLER                PICTURE IS X(20) VALUE
           "*** EDIT TOTALS ***".
           02 FILLER                PICTURE IS X(14) VALUE
           "RECORDS READ:".
           02 TRANSACTIONS-READ     PICTURE IS Z,ZZ9.
           02 FILLER                PICTURE IS X(61) VALUE SPACES.
           SKIP1
       01  SUMMARY-2.
           02 FILLER                PICTURE IS X(40) VALUE SPACES.
           02 FILLER                PICTURE IS X(21) VALUE
           "RECORDS WITH ERRORS:".
```

Page 7 of 18

```
           02 TRANSACTIONS-WITH-ERRORS
                                   PICTURE IS Z,ZZ9.
           02 FILLER               PICTURE IS X(54) VALUE SPACES.
           SKIP1
       01  SUMMARY-3.
           02 FILLER               PICTURE IS X(40) VALUE SPACES.
           02 FILLER               PICTURE IS X(25) VALUE
           "PCT RECORDS WITH ERRORS:".
           02 TRANSACTIONS-WITH-ERRORS
                                   PICTURE IS ZZ9.99.
           02 FILLER               PICTURE IS X(49) VALUE SPACES.
           SKIP1
       01  SUMMARY-4.
           02 FILLER               PICTURE IS X(40) VALUE SPACES.
           02 FILLER               PICTURE IS X(20) VALUE
           "TOTAL ERRORS FOUND:".
           02 ERRORS-FOUND         PICTURE IS Z,ZZ9.
           02 FILLER               PICTURE IS X(55) VALUE SPACES.
           SKIP1
       01  SUMMARY-5.
           02 FILLER               PICTURE IS X(40) VALUE SPACES.
           02 FILLER               PICTURE IS X(36) VALUE
           "AVE ERRORS IN RECORDS WITH ERRORS:".
           02 AVE-ERRORS-FOUND     PICTURE IS 9.99
```

Page 8 of 18

```
           02 FILLER                         PICTURE IS X(40) VALUE SPACES.
           SKIP2
      *
      *     ERROR MESSAGE RECORD DEFINITIONS
      *
       01  ERROR-MESSAGE-1.
           02 FILLER                         PICTURE IS X(30) VALUE SPACES.
           02 FILLER                         PICTURE IS X(90) VALUE
           "*** ERROR ***  INVENTORY NUMBER IS INVALID".
           SKIP1
       01  ERROR-MESSAGE-2.
           02 FILLER                         PICTURE IS X(30) VALUE SPACES.
           02 FILLER                         PICTURE IS X(90) VALUE
           "*** ERROR ***  TRANSACTION CODE NOT ADD OR DELETE".
           SKIP1
       01  ERROR-MESSAGE-3.
           02 FILLER                         PICTURE IS X(30) VALUE SPACES.
           02 FILLER                         PICTURE IS X(90) VALUE
           "*** ERROR ***  QUANTITY MISSING OR NOT NUMERIC".
           SKIP1
       01  ERROR-MESSAGE-4.
           02 FILLER                         PICTURE IS X(30) VALUE SPACES.
           02 FILLER                         PICTURE IS X(90) VALUE
           "*** ERROR ***  DESCRIPTION MISSING".
```

Figure 16.18
(continued)


```
           EJECT
       PROCEDURE DIVISION.
       MAIN-CONTROL SECTION.
           SKIP2
       M-C-001.
      *
      *    ****  BOTH FILES ARE OPENED, TITLES PRINTED,
      *          AND FIRST TRANSACTION READ.
      *
           OPEN INPUT TRANSACTION-FILE
                OUTPUT PRINT-FILE.
           PERFORM PRINT-TITLES.
           READ TRANSACTION-FILE
                INTO
                    INVENTORY-RECORD-INPUT
                AT END
                    MOVE SPACES TO INVENTORY-RECORD-INPUT.
           SKIP3
       M-C-002.
      *
      *    ****   THE PROCESS OF EDITING AND READING
      *           CONTINUES UNTIL ALL TRANSACTIONS
      *           HAVE BEEN PROCESSED.
      *
```



```
           PERFORM EDIT-PROCESS
                UNTIL
                    INVENTORY-RECORD-INPUT = SPACES.
           SKIP3
       M-C-003.
      *
      *    ****  PRINT FINAL STATISTICS, PRINT
      *          TRAILERS, CLOSE ALL FILES, AND
      *          TERMINATE EXECUTION.
      *
           PERFORM PRINT-CONTROL-TOTALS.
           PERFORM PRINT-TRAILERS.
           CLOSE TRANSACTION-FILE
                 PRINT-FILE.
           STOP RUN.
           SKIP3
           SKIP3
       EDIT-PROCESS SECTION.
           SKIP2
       E-P-001.
      *
      *    ****  IF PAGE OVERFLOW HAS OCCURRED,
      *          PRINT TRAILERS FOLLOWED BY
      *          TITLES.
```

Page 11 of 18

```
      *
           IF PAGE-OVERFLOW
               THEN
                   PERFORM PRINT-TRAILERS
                   PERFORM PRINT-TITLES.
           SKIP3
       E-P-002.
      *
      *    ****  PRINT THE RECENTLY READ TRANSACTION.
      *
           MOVE INVENTORY-RECORD-INPUT TO RECORD-IMAGE.
           ADD 1 TO TRANSACTIONS-READ OF SUBTOTALS.
           WRITE PRINT-RECORD
               FROM
                    INVENTORY-RECORD-OUTPUT
               AFTER ADVANCING
                    2 LINES.
           ADD 2 TO CURRENT-LINE.
           SKIP3
       E-P-003.
      *
      *    ****  PERFORM A VALIDITY CHECK ON EACH
      *          FIELD WITHIN THE TRANSACTION.
      *
```

Page 12 of 18

```
           IF INVENTORY-NUMBER NOT NUMERIC
               THEN
                    ADD 1 TO ERRORS-FOUND OF SUBTOTALS
                    ADD 1 TO CURRENT-LINE
                    MOVE 1 TO TRANSACTION-STATUS
                    WRITE PRINT-RECORD
                         FROM
                              ERROR-MESSAGE-1
                         AFTER ADVANCING
                              1 LINES.
           IF NOT VALID-TRANSACTION-CODE
               THEN
                    ADD 1 TO ERRORS-FOUND OF SUBTOTALS
                    ADD 1 TO CURRENT-LINE
                    MOVE 1 TO TRANSACTION-STATUS
                    WRITE PRINT-RECORD
                         FROM
                              ERROR-MESSAGE-2
                         AFTER ADVANCING
                              1 LINES.
           IF QUANTITY NOT NUMERIC
               THEN
                    ADD 1 TO ERRORS-FOUND OF SUBTOTALS
                    ADD 1 TO CURRENT-LINE
```

Figure 16.18
(continued)


```
                MOVE 1 TO TRANSACTION-STATUS
                WRITE PRINT-RECORD
                    FROM
                        ERROR-MESSAGE-3
                    AFTER ADVANCING
                        1 LINES.
            IF DESCRIPTION = SPACES
                THEN
                    ADD 1 TO ERRORS-FOUND OF SUBTOTALS
                    ADD 1 TO CURRENT-LINE
                    MOVE 1 TO TRANSACTION-STATUS
                    WRITE PRINT-RECORD
                        FROM
                            ERROR-MESSAGE-4
                        AFTER ADVANCING
                            1 LINES.
            SKIP3
       E-P-004.
      *
      *    ****  IF ANY FIELD IN THE TRANSACTION WAS
      *          FOUND IN ERROR, THEN THE TRANSACTION
      *          IS BAD.
      *
            IF TRANSACTION-IS-BAD
```



```
                THEN
                    ADD 1 TO TRANSACTIONS-WITH-ERRORS OF SUBTOTALS
                    MOVE 0 TO TRANSACTION-STATUS.
           SKIP3
       E-P-005.
      *
      *    ****  READ THE NEXT TRANSACTION.
      *
           READ TRANSACTION-FILE
               INTO
                   INVENTORY-RECORD-INPUT
               AT END
                   MOVE SPACES TO INVENTORY-RECORD-INPUT.
           SKIP3
           SKIP3
       PRINT-TITLES SECTION.
           SKIP1
       P-TI-001.
      *
      *    ****  PRINT THE TITLES THAT APPEAR AT THE
      *          TOP OF EACH PAGE.
      *
           WRITE PRINT-RECORD
               FROM
```

Page 15 of 18

```
                 TITLE-1
             AFTER ADVANCING
                 TO-TOP-OF-PAGE.
         WRITE PRINT-RECORD
             FROM
                 TITLE-2
             AFTER ADVANCING
                 1 LINES.
         WRITE PRINT-RECORD
             FROM
                 TITLE-3
             AFTER ADVANCING
                 1 LINES.
         WRITE PRINT-RECORD
             FROM
                 TITLE-4
             AFTER ADVANCING
                 2 LINES.
         MOVE 7 TO CURRENT-LINE.
         SKIP3
         SKIP3
     PRINT-TRAILERS SECTION.
         SKIP1
     P-TR-001.
```

Page 16 of 18

```
      *
      *  ***     INCREMENT THE PAGE NUMBER AND PRINT THE
      *          TRAILERS THAT APPEAR AT THE BOTTOM OF
      *          EACH PAGE.
      *
         ADD 1 TO PAGE-NUMBER OF SUBTOTALS.
         MOVE CORRESPONDING SUBTOTALS TO
                                 TRAILER-2.
         WRITE PRINT-RECORD
             FROM
                 TRAILER-1
             AFTER ADVANCING
                 TO-BOTTOM-OF-PAGE.
         WRITE PRINT-RECORD
             FROM
                 TRAILER-2
             AFTER ADVANCING
                 1 LINES.
         SKIP3
         SKIP3
     PRINT-CONTROL-TOTALS SECTION.
         SKIP1
     P-C-T-001.
      *
```

Figure 16.18
(continued)


```
      *    ****   COMPUTE AND PRINT THE STATISTICS THAT
      *           APPEAR THE REPORT'S END.

           MOVE TRANSACTIONS-READ OF SUBTOTALS TO
                TRANSACTIONS-READ OF SUMMARY-1.
           WRITE PRINT-RECORD
               FROM
                   SUMMARY-1
               AFTER ADVANCING
                   1 LINES.
           SKIP1
           MOVE TRANSACTIONS-WITH-ERRORS OF SUBTOTALS TO
                TRANSACTIONS-WITH-ERRORS OF SUMMARY-2.
           WRITE PRINT-RECORD
               FROM
                   SUMMARY-2
               AFTER ADVANCING
                   1 LINES.
           SKIP1
           COMPUTE TRANSACTIONS-WITH-ERRORS
                              OF SUMMARY-3 = TRANSACTIONS-WITH-ERRORS
                                             OF SUBTOTALS
                                                   /
                                            TRANSACTIONS-READ
```



```
                                             OF SUBTOTALS * 100.
           WRITE PRINT-RECORD
               FROM
                   SUMMARY-3
               AFTER ADVANCING
                   1 LINES.
           SKIP1
           MOVE ERRORS-FOUND OF SUBTOTALS TO
                ERRORS-FOUND OF SUMMARY-4.
           WRITE PRINT-RECORD
               FROM
                   SUMMARY-4
               AFTER ADVANCING
                   1 LINES.
           SKIP1
           COMPUTE AVE-ERRORS-FOUND = ERRORS-FOUND OF SUBTOTALS /
                                 TRANSACTIONS-WITH-ERRORS
                                      OF SUBTOTALS.
           WRITE PRINT-RECORD
               FROM
                   SUMMARY-5
               AFTER ADVANCING
                   1 LINES.
```

Questions for Review

1. Why were input files required to contain a blank record as the last record in the file if the file was to be processed using the ACCEPT verb? What would happen if the blank record was omitted entirely? If it was the first record in the file?
2. Using the DISPLAY instruction, what steps would you have to take to guarantee that report titles actually appeared at the top of each page of paper stock?
3. What is the advantage to using low volume I/O statements rather than conventional file structures? The disadvantage?
4. What are buffers? Can an input record be processed directly in a buffer, or must the buffer's contents be transferred to the WORKING-STORAGE SECTION first? To an output record? Can a report title or trailer be defined and assigned an initial value within a buffer?
5. What is the role of the SPECIAL-NAMES paragraph where conventional file structures are concerned?

Problems

1. Examine the following segment of a COBOL program. What, if anything, is wrong?

```
INPUT OUTPUT SECTION.
FILE CONTROL
    SELECT TRANSACTION-FILE ASSIGN TO 2540R-SYSIN.
```

2. What, if anything, is wrong with the following segment of a COBOL program?

```
INPUT-OUTPUT SECTION.
FILE CONTROL.
    SELECT TRANSACTION-FILE ASSIGN TO UR-2540R-S-SYSIN.
    SELECT PRINT-FILE ASSIGN TO UR-1403-S-SYSPRINT.
DATA DIVISION.
FD  TRANSACTION.
01  TRANSACTION-RECORD    PICTURE IS X(120).
FD. PRINTER-FILE.
01  PRINT-RECORD.
```

3. Examine the following COBOL instructions. What, if anything, is wrong?

```
READ TRANSACTION-FILE
    AT END
       MOVE "YES" TO FILE-STATUS.
    INTO
       INVENTORY-RECORD-INPUT.
WRITE PRINT-FILE
    AFTER ADVANCING
       1 LINE
    FROM
       INVENTORY-RECORD-OUTPUT.
```

4. Examine the following FILE SECTION from a COBOL program. What, if anything, is wrong?

```
DATA DIVISION.
FILE SECTION.
FD  PRINT-FILE
    LABEL RECORDS ARE OMITTED.
01  TITLE-1.
    02 FILLER           PICTURE IS X(53) VALUE SPACES.
    02 FILLER           PICTURE IS X(67) VALUE
    "COX INDUSTRIES".
01  TITLE-2.
    02 FILLER           PICTURE IS X(52) VALUE SPACES.
    02 FILLER           PICTURE IS X(68) VALUE
    "INVENTORY SYSTEM".
01  TITLE-3.
    02 FILLER           PICTURE IS X(54) VALUE SPACES.
    02 FILLER           PICTURE IS X(66) VALUE
    "EDIT REPORT".
01  TITLE-4.
    02 FILLER           PICTURE IS X(20) VALUE SPACES.
    02 FILLER           PICTURE IS X(40) VALUE
    ".........1.........2.........3.........4".
    02 FILLER           PICTURE IS X(60) VALUE
    ".........5.........6.........7.........8".
01  BLANK-LINE          PICTURE IS X(120) VALUE SPACES.
01  TRAILER-1.
    02 FILLER           PICTURE IS X(20) VALUE SPACES.
    02 FILLER           PICTURE IS X(40) VALUE
    ".........1.........2.........3.........4".
    02 FILLER           PICTURE IS X(60) VALUE
    "........5.........6.........7.........8".
01  TRAILER-2.
    02 FILLER           PICTURE IS X(55) VALUE SPACES.
    02 FILLER           PICTURE IS X(05) VALUE
    "PAGE-".
    02 PAGE-NUMBER      PICTURE IS Z,ZZ9.
    02 FILLER           PICTURE IS X(55) VALUE SPACES.
```

Programming Assignments

1. Revise the Thomas Information Systems program from chapter 11 so that all of the ACCEPT and DISPLAY statements have been replaced by READ and WRITE statements, respectively.
2. Revise the SCAGs program from chapter 13 so that all of the ACCEPT and DISPLAY statements have been replaced by READ and WRITE statements, respectively.

Table Definition and Processing
Part 1

17

Chapter Outline

The program developed in chapter 13 examined time cards used to produce paychecks in a payroll application. That edit program inspected each and every field in every time card to determine whether one or more of those fields were in error. One such field was hourly rate.

Many organizations have a fixed table of hourly salaries. The existence of this table means every employee is being paid an hourly rate found in the table. Promotions and increases in pay really involve going from one entry in the table of hourly rates to yet another (i.e., a larger one).

Organizations utilizing tables of hourly salaries might not place an employee's actual hourly rate on his or her time card. It is quite possible that the time cards will contain the number of the entry in the table of hourly rates appropriate to that person's salary. If entry numbers aren't used, it is also possible that symbolic names have been assigned for each entry in the table. Examine the table of hourly rates in figure 17.1. This table is the hourly rate guide used by the payroll office of Frodo Kay Cosmetics, Inc. Frodo Kay Cosmetics employs a rather large work force to support its network of self-employed sales representatives. Each representative sells cosmetics and related items at home-cosmetic parties. Orders for cosmetics are recorded by the sales representative at each party. The orders are subsequently telephoned to the district office where the items are picked from the inventory and delivered by car to the representative who placed the order.

Frodo Kay Cosmetics employs college students to fill orders in the warehouse as well as to deliver orders. The entries in the rate table for these individuals are symbolically termed *Day Labor*. The four categories of day laborers are DL-1, DL-2, DL-3, and DL-4. All employees hired as day laborers start at the lowest rate, in table entry number 1, in category DL-1. As the length of employment continues, these employees are promoted from $3.10 per hour to a maximum of $4.35 per hour. That is, they go from table entry number 1, or DL-1, up to table entry number 4, or DL-4.

Particularly promising day laborers are eligible for promotion to supervisory status. These supervisors oversee the work of the day laborers. The two categories of supervisors are represented by table entry numbers 5 and 6, also known as SL-1 and SL-2. Similar categories exist for office workers (OW-1 to OW-3) and supervisory office workers (SO-1 and SO-2). The flexibility that a table of hourly rates provides is quite impressive. Consider the issue of cost of living increases.

Each year the owners of Frodo Kay Cosmetics upgrade all hourly rates paid to its employees. Frodo Kay Cosmetics realizes that the firm is responsible for providing rates of pay comparable to similar positions in other companies in their geographic region. Because a table of rates is used, an overall increase of five percent, for instance, can be calculated for each entry in the table. The revised table can now be used for

Figure 17.1
Table of hourly rates

Entry number	Symbolic name	Hourly rate
1	DL-1	3.10
2	DL-2	3.45
3	DL-3	3.75
4	DL-4	4.35
5	SL-1	5.10
6	SL-2	6.26
7	OW-1	3.66
8	OW-2	4.88
9	OW-3	5.25
10	SO-1	6.55
11	SO-2	7.85

payroll purposes. Since each time card contains only the symbolic pay category of the employee (e.g., DL-2), an increase in the rate corresponding to that payroll category does not require a change in the time card. After all, the employee hasn't changed payroll categories. What has changed is the rate of compensation for all employees in that category. The time card contains the same symbolic pay category after the overall increase has been calculated as it did prior to the revision in the salary structure.

Discontinuous Value Ranges in Variables

The edit program that processes time cards must now be structured to not expect an actual hourly rate on the time card. Instead, each time card will contain a four-character symbolic payroll category. For Frodo Kay Cosmetics, this means each time card can contain one of eleven possible symbolic payroll categories to be valid. A time card containing a payroll category that is not among the eleven shown in the earlier table is in error.

The COBOL coding in figure 17.2 could be used to examine the field named PAYROLL-CATEGORY. As you can see, the IF statement is quite long. What if the table of hourly rates contained 20, 100, or even more rate categories? Clearly, as the size of the table increases, the less likely it is that a single IF statement can be used to verify the PAYROLL-CATEGORY.

Concepts, Definition, and Initialization of Tables

A method exists within COBOL to define and process other tables much like the hourly rate table. To create tables in COBOL requires the introduction of *multi-valued variables.*

Multi-valued Variables
Variables that can hold, simultaneously, more than one value. Also known as vectors, tables, and/or matrices.

All of the variables used so far have been single-valued variables. That is, at any given point in time a single-valued variable could hold one value. Multi-valued variables, however, are capable of containing many different values at the same time.

```
IF PAYROLL-CATEGORY OF TIME-CARD-INPUT NOT = "DL-1" AND
   PAYROLL-CATEGORY OF TIME-CARD-INPUT NOT = "DL-2" AND
   PAYROLL-CATEGORY OF TIME-CARD-INPUT NOT = "DL-3" AND
                   .
                   .
                   .
   PAYROLL-CATEGORY OF TIME-CARD-INPUT NOT = "OW-2" AND
   PAYROLL-CATEGORY OF TIME-CARD-INPUT NOT = "OW-3" AND
                   .
                   .
                   .
   PAYROLL-CATEGORY OF TIME-CARD-INPUT NOT = "SL-2"
    THEN
          ADD 1 TO ERRORS-FOUND OF SUBTOTALS
          ADD 1 TO CURRENT-LINE
          MOVE "YES" TO TRANSACTION-STATUS
          WRITE PRINT-RECORD
              FROM
                  ERROR-MESSAGE-4
              AFTER ADVANCING
                  1 LINES.
```

Figure 17.2
Separate IF-THEN statements to check PAYROLL-CATEGORY

Figure 17.3
Constructing a table of hourly rates

```
01  HOURLY-RATE-TABLE.
    02 EACH-ENTRY                 OCCURS 11 TIMES.
       03 PAYROLL-CATEGORY        PICTURE IS X(04).
       03 HOURLY-RATE             PICTURE IS 9V99.
```

Definition

Examine the definition of an hourly rate table in COBOL shown in figure 17.3. If the 02-level definition were removed, this would appear to be a seven-position record description. The presence of the 02 specifies that this seven-position entity is to be repeated eleven times. That is, the OCCURS entry converts a record description into a table with (eleven) rows. Each row is seven positions long. The total number of storage locations required by this table is seventy-seven (11 × 7) characters.

Statement Format

```
8/12
↓
levelnumber  repeateditem  OCCURS n TIMES
                           [PICTURE IS attributes].
```

This table definition appears in the WORKING-STORAGE SECTION of our COBOL edit program. Like all definitions that begin with an 01 level, the 01 appears in the A-Margin, while everything else resides in the B-Margin.

Each of the 03-level items indicates how every row is to be organized. In this instance, each of the eleven rows contains two variables. The first variable provides the symbolic name assigned to the hourly rate the row holds—for example DL-1. The second variable further describes that row by specifying the actual dollar-cent numeric value of the hourly wage. The first row in the table represents the hourly rate for the lowest level of day laborer. Thus, the first variable in the row, PAYROLL-CATEGORY, is assigned the value DL-1. The second variable in the row, HOURLY-RATE, is assigned the numeric value of 3.10 (decimal point added for clarity).

Initialization

The table in its present form contains no data. We know what each row should contain. The question is: How can we cause the HOURLY-RATE-TABLE to contain the values shown for the hourly payroll table?

Multi-valued variables cannot be assigned initial values like single-valued variables can. The VALUE clause cannot be used in any variable definition that is subordinate to an OCCURS clause. An example of what is permitted is illustrated in figure 17.4. Has the table grown so that more than seventy-seven characters are involved? No. The table remains the same size as before. DATA-PORTION defines a string of seventy-seven characters that come from concatenating each of the rows from the payroll table one after the other. The phrase EACH-ENTRY REDEFINES DATA-PORTION causes the data to occupy the very same storage locations in computer memory as in the table. As a result, the table now contains its data.

```
       01  HOURLY-RATE-TABLE.
           02 DATA-PORTION             PICTURE IS X(77) VALUE
           "DL-1310DL-2345DL-3375DL-4435SL-1510SL-2626OW-1366OW-2488OW-3
      -    "5255O-1655SO-2785".
           02 EACH-ENTRY               REDEFINES DATA-PORTION
                                       OCCURS 11 TIMES.
              03 PAYROLL-CATEGORY      PICTURE IS X(04).
              03 HOURLY-RATE           PICTURE IS 9V99.
```

Figure 17.4
Placing data into tables

```
           IF PAYROLL-CATEGORY OF TIME-CARD-INPUT NOT =
              PAYROLL-CATEGORY OF HOURLY-RATE-TABLE
                   THEN
                       ADD 1 TO ERRORS-FOUND OF SUBTOTALS
                       ADD 1 TO CURRENT-LINE
                       MOVE "YES" TO TRANSACTION-STATUS
                       WRITE PRINT-RECORD
                            FROM
                                ERROR-MESSAGE-4
                            AFTER ADVANCING
                                1 LINES.
```

Figure 17.5
Incorrect reference to a multi-valued variable

Table Processing

The table in our edit program would permit the use of an IF statement that is much shorter than the one shown in figure 17.2. You might be tempted to construct the shortened version as shown in figure 17.5. Although this particular IF statement is much shorter, it contains an unacceptable reference to the multi-valued variable PAYROLL-CATEGORY OF HOURLY-RATE-TABLE.

PAYROLL-CATEGORY OF TIME-CARD-INPUT is a single-valued variable. When a time card record is read and its contents are placed in the WORKING-STORAGE SECTION in a record description named TIME-CARD-INPUT, a variable within TIME-CARD-INPUT called PAYROLL-CATEGORY contains a single four-character symbolic name for the payroll category to which the employee belongs. The variable PAYROLL-CATEGORY OF HOURLY-RATE-TABLE is, on the other hand, a multi-valued variable. This multi-valued variable holds eleven values. Consequently, the most recent IF statement would be rejected because the COBOL compiler could not determine which of the eleven values in PAYROLL-CATEGORY OF HOURLY-RATE-TABLE you were attempting to reference.

The Need for an Iterative Structure

Iterative Structure
The ability, when processing tables, to successively examine each entry in a multi-valued variable.

Subscript
A numeric variable or literal that specifies the single entry within a multi-valued variable that we wish to examine or alter. The item is usually identified by its row-column number.

In order to use a multi-valued variable like PAYROLL-CATEGORY OF HOURLY-RATE-TABLE in COBOL programs, we must structure an *iterative,* or looping, function that gives access one at a time to each of the many values such a variable contains. Inspect the COBOL segments shown in figure 17.6.

Figure 17.6 illustrates table processing through use of a *subscript.* A subscript is a numeric variable, or literal, that tells the computer which single value of a multi-valued variable the program wishes to access. A variable named ROW-POINTER is defined in the WORKING-STORAGE SECTION to serve as a subscript. The value in ROW-POINTER specifies which row in HOURLY-RATE-TABLE the program is accessing at any given point in time. If ROW-POINTER contains the value 03, then our program is accessing the third row of the HOURLY-RATE-TABLE. This is the row for symbolic payroll category DL-3 with a corresponding hourly wage of $3.75 per hour.

To determine if PAYROLL-CATEGORY OF TIME-CARD-INPUT contains a valid payroll designation requires the four-character designation to be compared successively to each row in the table. To access the first row, ROW-POINTER must contain 01, the second 02, the third 03, and so on. Because ROW-POINTER is going to be incremented from one to eleven by one, we must first set it to zero—an action accomplished by the first MOVE instruction. As a result, the first time the instruction ADD 1 TO ROW-POINTER is executed, ROW-POINTER increases in value from zero to one.

The first IF statement compares the payroll designation from the time card to that in the first row of the hourly rate table. Multi-valued variables must be followed by what is known as a subscript specification. The subscript specifies which of the eleven rows we are comparing. Obviously, since ROW-POINTER contains the value of 01, the IF statement is comparing the value from the time card to that of the first row of the table. Note that the subscript must be enclosed in parentheses. Subscripts can be numeric variables or numeric literals.

If the value from the time card matches the first row of the table, the condition is true and the statement associated with the THEN portion of that IF statement will be executed. That statement is known as unconditional transfer of control; otherwise known as a GO TO statement.

A GO TO statement causes the computer to go immediately to the paragraph in the PROCEDURE DIVISION whose name follows the phrase *GO TO.* In this case control proceeds directly from the GO TO instruction to the instruction immediately following the paragraph E-P-004. Supposedly, E-P-004 marks the end of the testing of PAYROLL-CATEGORY OF TIME-CARD-INPUT and the beginning of the testing of the next field in that record description. If a match is found, the value on the time card is valid. If the value is valid, it is unnecessary to examine the other rows of the table for a possible match, since only one such match is possible and it has already been found.

Assume, however, that the value in the time card does not match the value in the first row of the table. In such an instance, control proceeds to the second IF statement in the COBOL segment. This second IF statement inquires as to the value held in ROW-POINTER. At this point the value held is 01. Only the first row of the table has been examined and no match was found. Since 01 is not eleven, then at least one more row can be examined for a possible match. That being the case, control is automatically directed to the paragraph E-P-003 by the GO TO statement. Here the value in ROW-POINTER is incremented again by one. ROW-POINTER now contains 02 rather than 01. Consequently, the value in the time card will be compared to the value in the second row of the table, and the process just described occurs one more time.

Figure 17.6
Proper reference of multi-valued variables

Page 1 of 2

```
       01  HOURLY-RATE-TABLE.
           02 DATA-PORTION            PICTURE IS X(77) VALUE
           "DL-1310DL-2345DL-3375DL-4435SL-1510SL-2626OW-1366OW-2488OW-3
      -    "525SO-1655SO-2785".
           02 EACH-ENTRY              REDEFINES DATA-PORTION
                                      OCCURS 11 TIMES.
              03 PAYROLL-CATEGORY     PICTURE IS X(04).
              03 HOURLY-RATE          PICTURE IS 9V99.
           .
           .
       01  ROW-POINTER                PICTURE IS 9(02).
           .
           .
           MOVE 0 TO ROW-POINTER.
       E-P-003.
      *
      *      THIS SEGMENT CHECKS THE VALUE FOR PAYROLL
      *      CATEGORY FROM THE TIMECARD WITH EACH VALUE
      *      IN THE HOURLY RATE TABLE.  IF NO MATCH IS
      *      FOUND, THEN THE TIMECARD CONTAINS AN ERROR.
      *
           ADD 1 TO ROW-POINTER.
```

Page 2 of 2

```
           IF PAYROLL-CATEGORY OF TIME-CARD-INPUT =
               PAYROLL-CATEGORY OF HOURLY-RATE-TABLE (ROW-POINTER)
                 THEN
                     GO TO E-P-004.
           IF ROW-POINTER NOT = 11
                 THEN
                     GO TO E-P-003.
           ADD 1 TO ERRORS-FOUND OF SUBTOTALS.
           ADD 1 TO CURRENT-LINE.
           MOVE "YES" TO TRANSACTION-STATUS.
           WRITE PRINT-RECORD
                 FROM
                     ERROR-MESSAGE-4
                 AFTER ADVANCING
                     1 LINES.
       E-P-004.
                 .
                 .
                 .
```

Assume the time card contains an illegal value in PAYROLL-CATEGORY. This means no match will be found in the table. As a result, this whole process will occur eleven times—once for each row possible. On the eleventh iteration, when no match is found in the first IF statement, control will pass to the second IF statement. Because ROW-POINTER already contains eleven, no more rows can be checked. If no more rows can be checked, then no match was found. If no match was found, an error exists. If an error exists, we need to increment error counts, set bi-stable error switches, and print an error message. Hence, control is not sent back to E-P-003 where the iterative process could occur one more time. Instead, control passes to the instructions at the end of E-P-003 that accomplish the error-processing tasks just noted.

Figure 17.7
Erroneous table-processing logic

```
       E-P-003.
      *
      *        THIS SEGMENT CHECKS THE VALUE FOR PAYROLL
      *        CATEGORY FROM THE TIMECARD WITH EACH VALUE
      *        IN THE HOURLY RATE TABLE.  IF NO MATCH IS
      *        FOUND, THEN THE TIMECARD CONTAINS AN ERROR.
      *
           IF PAYROLL-CATEGORY OF TIME-CARD-INPUT =
               PAYROLL-CATEGORY
                           OF HOURLY-RATE-TABLE (ROW-POINTER)
               THEN
                   GO TO E-P-003.
           IF ROW-POINTER NOT = 11
               THEN
                   GO TO E-P-004.
           ADD 1 TO ERRORS-FOUND OF SUBTOTALS.
           ADD 1 TO CURRENT-LINE.
           MOVE "YES" TO TRANSACTION-STATUS.
           WRITE PRINT-RECORD
               FROM
                   ERROR-MESSAGE-4
               AFTER ADVANCING
                   1 LINES.
       E-P-004.
```

To Err Is Very Simple

Table processing in COBOL is the cause of problems for many beginning programmers. Why? It is easy to commit an error in logic that can produce some very strange results. Carefully examine a slightly modified table-processing segment in COBOL shown in figure 17.7. Do you notice anything wrong?

The revised COBOL logic is missing two very important statements. These statements were purposely omitted. However, a programmer, particularly a beginner, might unintentionally remove source statements during the program debugging process.

The first statement omitted is the MOVE instruction that assigns ROW-POINTER a numeric value that makes the incrementing of this subscript possible. Consequently, ROW-POINTER contains no initial value, so when ROW-POINTER is accessed in the first IF statement, the COBOL program will be abnormally terminated by the computer system. The termination was abnormal, and program execution prematurely halted, because the program directed the computer to access a row in the table but neglected to specify a numeric value in the range of 01 to 11.

If the MOVE instruction were replaced, the program would still abnormally terminate because (1) the ADD instruction that increments the subscript is missing; (2) the subscript would contain the value of 00 in the first IF statement; (3) zero is not a row number; (4) the first row in the table is row 01; and (5) you cannot access a row that does not exist.

Other problems are possible, too. The problem with the GO TO instruction is that it is very easy to specify the wrong paragraph name. To do so disrupts the intended logical structure of the program. Did you notice that this is the case here? If a match is found, control goes to the wrong place. If no match is found, control is again misdirected. It is very easy to make such an error and not realize you have done so.

Statement Format

```
12
↓
GO TO paragraphname.
```

An Alternative—the PERFORM

It might have occurred to you that not only is it easy to incorrectly specify the COBOL statements to process a table, but that a significant number of statements are required to process a table correctly. A shorter alternative does exist, however. Remember, table processing was introduced as a means of shortening COBOL programs. This shorter alternative involves a familiar instruction—PERFORM.

Examine the COBOL segments to search the hourly rate table (figure 17.8) for a match with the payroll designation from the time card. You may still be unimpressed by the relatively large number of statements required to process the hourly rate table. It might even appear easier to construct a single IF statement with eleven separate conditions than to construct the iterative processing logic shown. This may be true for tables containing as few as 11 entries, but what of tables containing 200 entries? Clearly, it is easier to construct iterative processing logic than to write a single IF statement with 200 separate conditions!

The PERFORM statement has been used in the MAIN-CONTROL SECTIONS of our COBOL report and edit programs to control the iterative processing of records in the input file. That is, records in the input file were continually read and processed until the file was exhausted. So, the UNTIL option is very familiar—the iterative process continues until the termination specification, ROW-POINTER = 11, is satisfied.

The VARYING option of the PERFORM statement is very powerful. VARYING ROW-POINTER FROM 1 BY 1 first assigns ROW-POINTER an initial value of 01, and second, automatically increments the value in ROW-POINTER by one. Consequently, the VARYING option eliminates the need for the ADD instruction that incremented ROW-POINTER in our earlier version of the iterative processing logic. What about the MOVE instruction that initialized ROW-POINTER to zero?

In figure 17.6, the MOVE instruction that assigned a valid initial value to the subscript ROW-POINTER was a means to avoid a possible abnormal termination. If ROW-POINTER did not contain an appropriate numeric value at the start of the search, the following ADD instruction would not operate properly. With the PERFORM instruction this is no longer a concern. The PERFORM not only properly increments the subscript and terminates the loop, but it also sets the subscript with the correct starting value (FROM). It seems that the PERFORM is a fail-safe instruction. Well, not quite.

The PERFORM statement shown in figure 17.8 is incorrect. The termination criterion (i.e., UNTIL) states that the loop should continue UNTIL ROW-POINTER = 11. The problem here is quite simple. If the condition in the UNTIL is false, control goes to SEARCH-TABLE. If the condition is true, control leaves the PERFORM for the next verb and SEARCH-TABLE is not executed. So, as constructed, the PERFORM instruction in figure 17.8 will only search the first ten rows in the table. Row 11 is never searched. Correcting this problem requires changing the UNTIL condition to UNTIL ROW-POINTER > 11 or UNTIL ROW-POINTER = 12. Either will work quite well.

Statement Format

```
12
↓
PERFORM sectionname
   VARYING subscript FROM startvalue BY increment
   UNTIL terminatingcondition.
```

Another advantage of the PERFORM verb is that control is automatically directed through each phase of the iterative process without the use of the GO TO statement.

Figure 17.8
Searching tables using the PERFORM instruction


```
       01  HOURLY-RATE-TABLE.
           02 DATA-PORTION            PICTURE IS X(77) VALUE
           "DL-1310DL-2345DL-3375DL-4435SL-1510SL-2626OW-1366OW-2488OW-3
      -    "525SO-1655SO-2785".
           02 EACH-ENTRY              REDEFINES DATA-PORTION
                                      OCCURS 11 TIMES.
               03 PAYROLL-CATEGORY    PICTURE IS X(04).
               03 HOURLY-RATE         PICTURE IS 9V99.
             .
             .
       01  ROW-POINTER                PICTURE IS 9(02).
             .
             .
       01  MATCH-FOUND                PICTURE IS X(03) VALUE "NO".
           88 NO-MATCH-FOUND                        VALUE "NO".
                     .
                     .
       E-P-003.
      *
      *      THIS SEGMENT CHECKS THE VALUE FOR PAYROLL
      *      CATEGORY FROM THE TIMECARD WITH EACH VALUE
```



```
      *      IN THE HOURLY RATE TABLE.  IF NO MATCH IS
      *      FOUND, THEN THE TIMECARD CONTAINS AN ERROR.
      *
           MOVE 0 TO ROW-POINTER.
           MOVE "NO" TO MATCH-FOUND.
           PERFORM SEARCH-TABLE
               VARYING
                    ROW-POINTER
               FROM 1 BY 1
               UNTIL
                    ROW-POINTER = 11.
           IF NO-MATCH-FOUND
               THEN
                    ADD 1 TO ERRORS-FOUND OF SUBTOTALS
                    ADD 1 TO CURRENT-LINE
                    MOVE "YES" TO TRANSACTION-STATUS
                    WRITE PRINT-RECORD
                        FROM
                            ERROR-MESSAGE-4
                        AFTER ADVANCING
                            1 LINES.
       E-P-004.
             .
```



```
             .
       SEARCH-TABLE SECTION.
       S-T-001.
      *
      *    THIS SECTION IS ENTERED EVERY TIME ROW POINTER IS
      *    INCREMENTED BY 1 IN THE PERFORM STATEMENT.
      *    IF A MATCH IS FOUND, THEN A BI-STABLE VARIABLE IS
      *    SET TO SHOW A MATCH EXISTS.  OTHERWISE THE BI-STABLE
      *    VARIABLE REMAINS "OFF".
      *
           IF PAYROLL-CATEGORY OF TIME-CARD-INPUT =
               PAYROLL-CATEGORY OF HOURLY-RATE-TABLE (ROW-POINTER)
                 THEN
                     MOVE "YES" TO MATCH-FOUND.
```

```
       01  HOURLY-RATE-TABLE.
           02 DATA-PORTION              PICTURE IS X(77) VALUE
           "DL-1310DL-2345DL-3375DL-4435SL-1510SL-2626OW-1366OW-2488OW-3
      -    "525SO-1655SO-2785".
           02 EACH-ENTRY                REDEFINES DATA-PORTION
                                        OCCURS 11 TIMES
                                        INDEXED BY ROW-POINTER.
              03 PAYROLL-CATEGORY       PICTURE IS X(04).
              03 HOURLY-RATE            PICTURE IS 9V99.
               .
               .
               .
       E-P-003.
      *
      *       THIS SEGMENT CHECKS THE VALUE FOR PAYROLL
      *       CATEGORY FROM THE TIMECARD WITH EACH VALUE
      *       IN THE HOURLY RATE TABLE.  IF NO MATCH IS
      *       FOUND, THEN THE TIMECARD CONTAINS AN ERROR.
      *
           SET ROW-POINTER TO 1.
           SEARCH EACH-ENTRY OF HOURLY-RATE-TABLE
                  AT END
                         ADD 1 TO ERRORS-FOUND OF SUBTOTALS
                         ADD 1 TO CURRENT-LINE
```



```
                         MOVE "YES" TO TRANSACTION-STATUS
                         WRITE PRINT-RECORD
                               FROM
                                      ERROR-MESSAGE-4
                               AFTER ADVANCING
                                      1 LINES
                  WHEN
                         PAYROLL-CATEGORY OF HOURLY-RATE-TABLE (ROW-POINTER) =
                         PAYROLL-CATEGORY OF TIMECARD-INPUT
                               NEXT SENTENCE.
       E-P-004.
```

Figure 17.9
Searching tables using the SEARCH instruction

Still Another Alternative—the SEARCH

Perhaps the best way to search a table for a desired item is by use of the SEARCH verb. This verb is a somewhat recent addition to the COBOL programming language. As a result, many, many COBOL programs in existence process tables without the use of the SEARCH verb. These programs use variants of the first two methods shown in table processing. Because program maintenance is a fact of life, it is necessary for beginning programmers to learn to process tables by using more than just the SEARCH verb. They must also be exposed to table processing using ADD, IF, and GO TO statements as well as the PERFORM statement.

The SEARCH verb offers the programmer the fewest number of statements required to examine a table, row by row, for a desired entry. Examine the COBOL segments in figure 17.9. Again, the segments are extracted from the time card edit program. The first item of interest concerns the definition of HOURLY-RATE-TABLE. The OCCURS clause has been modified to explain that a subscript for the table exists. That subscript is named ROW-POINTER.

Statement Format

```
12
↓
SEARCH repeateditem
   [AT END statement]
   WHEN condition.
```

Unlike earlier definitions of the table, where a separate definition for ROW-POINTER was necessary, the addition of the phrase *INDEXED BY* automatically causes ROW-POINTER to be defined. In fact, an additional declaration for ROW-POINTER would be considered a duplicate declaration by the COBOL compiler, causing an error message to be produced and preventing the program from being executed. Consequently, the COBOL compiler automatically allocates storage locations for variables named in the INDEXED BY clauses.

Statement Format

```
8/12
↓
levelnumber   repeateditem   OCCURS n TIMES
              INDEXED BY indexname
              [PICTURE IS attributes].
```

Like PERFORM, the SEARCH verb is an iterative instruction. The search proceeds from the first row specified to the last until the condition in the WHEN is satisfied. In this instance, the WHEN will not be true unless a match between a row and the time card is found. If such a match does exist, control will pass to the statement after the condition that must be fulfilled. Any statement can follow the WHEN condition. NEXT SENTENCE is used because the discovery of a match signifies the payroll designation is valid. Since it is valid, control should pass to paragraph E-P-004 where the next field in the time card will be examined.

The SEARCH instruction also contains an AT END clause. The statements following AT END can only become active if every row has been examined and no match was found. The WRITE instruction is the last instruction that is part of the AT END segment of the SEARCH. When the WRITE is completed, control leaves the SEARCH and, in this case, proceeds directly to the first executable statement in paragraph E-P-004.

Note that the entire SEARCH verb, with its associated AT END and WHEN segments, contains only one period—at the very end.

The SET verb must be executed prior to the SEARCH verb. SET specifies the row in which the search will begin. Why use SET rather than MOVE?

Statement Format

```
12
↓
SET indexname TO startvalue.
```

Addressing via Subscripts

Accessing variables in tables requires the use of subscripts to specify which of the many rows we want. These subscripts can be numeric literals or numeric variables. Obviously, the use of variables provides greater flexibility in constructing COBOL program logic. Variables can have their values modified during the course of program execution. Different values in subscripts that are numeric variables allow the same COBOL statement to access different rows in the table. Numeric literals, on the other hand, are static, unchanging.

Externally Formatted Variables

From the outset of this text all numeric variables have been externally formatted. This means that numeric variables held values in computer storage exactly as they appear when printed by a line printer or read by a card reader—in human-readable form.

We are accustomed to reading numeric values represented in decimal. Computers do not manipulate decimal values in printable, decimal form. These values must be converted to an internal (i.e., to the computer) format prior to use as subscripts. The conversion process uses the computer's instruction set to convert external decimal values to their internal, processable counterparts. These special conversion instructions are made a part of our COBOL programs automatically by the COBOL compiler. Because we are often unaware of these conversion instructions, we tend to be ignorant of their impact.

Note the COBOL variable definitions for ROW-POINTER in figure 17.10. The first of the two definitions looks like any definition of a numeric variable we have used thus far. The second definition is somewhat more unusual.

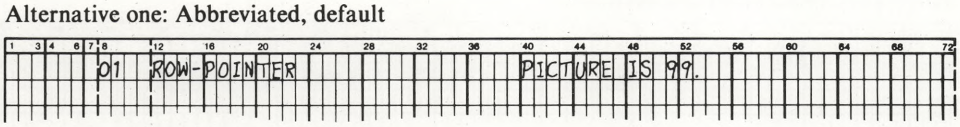
By specifying DISPLAY in the USAGE IS clause, the programmer informs the COBOL compiler that the variable must be defined so that its contents are in human-readable, external, decimal format. USAGE IS DISPLAY is a default specification. That is, every variable definition is assumed to be USAGE IS DISPLAY if the USAGE clause is omitted from the variable's declaration. Consequently, any reference to ROW-POINTER in a COBOL program requires the conversion of its value to a more readily processable form by the computer. Using ROW-POINTER in any arithmetic statement, MOVE statement, subscripting operation, or IF statement causes the COBOL compiler to generate the appropriate conversion instructions. Only the ACCEPT/READ and DISPLAY/WRITE instructions utilize numeric values in their human-readable form; hence, the use of the phrase USAGE IS DISPLAY.

Statement Format

```
8/12
↓
levelnumber   variablename   PICTURE IS attributes
              USAGE IS  ⎧DISPLAY          ⎫
                        ⎪COMPUTATIONAL    ⎪
                        ⎨COMPUTATIONAL-1  ⎬
                        ⎪COMPUTATIONAL-2  ⎪
                        ⎩COMPUTATIONAL-3  ⎭
```

These conversion operations are quite costly in terms of computer resources. They make programs larger and more time consuming to execute because the program contains more instructions than internally formatted variables require.

Alternative one: Abbreviated, default



Alternative two: Fully specified, confirms default

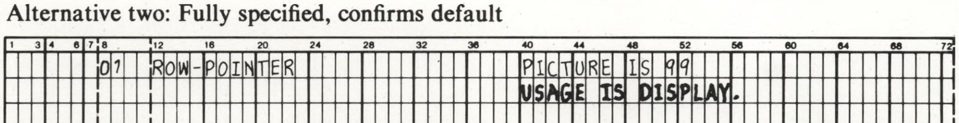


Figure 17.10
Fully specified variable definition—USAGE IS

Internally Formatted Variables

Numeric variables that are not the objects of ACCEPT/READ and DISPLAY/WRITE instructions are excellent candidates for definition as internally formatted. Variables to be printed must be in USAGE IS DISPLAY form or unintelligible characters will appear on the printed output. Variables to be given values with an ACCEPT/READ instruction must also be externally formatted. The transcription process records numeric values from source documents in external, decimal form.

Numeric variables that will never be involved in input/output operations can be defined as being internally formatted. To specify the internal attribute, the programmer must construct variable definitions that include the USAGE IS clause. Rather than USAGE IS DISPLAY, however, the choices are:

1. USAGE IS COMPUTATIONAL—fixed point binary
2. USAGE IS COMPUTATIONAL-1—floating point binary, short
3. USAGE IS COMPUTATIONAL-2—floating point binary, long
4. USAGE IS COMPUTATIONAL-3—fixed point decimal

All four alternatives provide numeric variables whose values are directly processable by the computer.[1] Because no conversion instructions are required, the program is more compact and executes more rapidly.

COMPUTATIONAL specifies that the numeric variable is to be represented as fixed decimal point, binary. Fixed decimal point means that the number of digits to the left and right of the decimal point can never vary. Binary means the variable's value will be recorded in binary.

COMPUTATIONAL-1 and COMPUTATIONAL-2 specifications also provide binary representation. Variables with either of the two specifications will feature floating point, or scientific notation, treatment of the decimal point. Since COMPUTATIONAL-1 and COMPUTATIONAL-2 variables are not used as subscripts, they will not be mentioned further in this particular chapter.

Like COMPUTATIONAL, COMPUTATIONAL-3 features fixed-point treatment of the decimal point. Unlike COMPUTATIONAL, however, the value in the variable is represented in an internal decimal. Internal decimal values can be manipulated directly without the need for conversion beforehand.

Two possible alternatives for the definition of ROW-POINTER are shown in figure 17.11. Both alternatives provide identical results.

Figure 17.11
Alternatives in internally formatted subscripts

Alternative one: Fixed point binary

```
       01  ROW-POINTER                 PICTURE IS 99
                                       USAGE IS COMPUTATIONAL.
```

Alternative two: Fixed point decimal

```
       01  ROW-POINTER                 PICTURE IS 99
                                       USAGE IS COMPUTATIONAL-3.
```

Relative Row Addressing

Whether USAGE IS DISPLAY, COMPUTATIONAL, or COMPUTATIONAL-3, the COBOL compiler must generate what is known as a storage-mapping function as a part of our COBOL program. A storage-mapping function converts a row-number specification into an address in computer memory where the value of a multi-valued variable resides. Storage mapping is basically a series of arithmetic instructions. The number of instructions involved depends on the complexity of the table being processed. Simple tables with few variables result in simple storage-mapping functions. These functions are required because we specify which row we desire to access by its row number, which is not an address in computer storage. It is a row specification relative to the starting address of a table in computer storage. What is needed is the actual memory address of the table—not the number of the row in the table.

Absolute Addressing

An alternative to relative row addressing features more rapid access to the entries in a table using storage locations directly rather than using row numbers. The SEARCH verb utilizes absolute table addressing.

The SEARCH verb requires that the OCCURS clause be immediately followed by an INDEXED BY clause. INDEXED BY names a subscript variable that will automatically be defined by the COBOL compiler. Subscripts defined in this manner are said to be index variables. Index variables contain the addresses of rows rather than row numbers which must be converted into addresses. The SET instruction specifies the address of the row at which a search operation must occur. Thus, the instruction SET ROW-POINTER TO 1 causes the address of row 1 in the table being searched to be placed into ROW-POINTER. Because all table references are carried out using addresses, the COBOL compiler does not need instructions to convert row numbers to addresses.

Additional Discussion of Table Processing

So far our only use of tables has been to access them to find out whether a desired value resides within them. We have used these tables in a read only mode of access. Tables can be used to store, or insert, values as well.

Storing Data in Tables

Assume the designers of the time card edit program desire to know how many time cards contain errors in the first field of each time card, the second, and so on. We might construct an error-frequency table that notes the frequency of error in each of the fields we are processing. Such a table might appear as shown in figure 17.12. ERROR-COUNT is a multi-valued variable with four different values possible at any given point in time. They are:

ERROR-COUNT (1)—The number of time cards with their first field in error.
ERROR-COUNT (2)—The number of time cards with their second field in error.
ERROR-COUNT (3)—The number of time cards with their third field in error.
ERROR-COUNT (4)—The number of time cards with their fourth field in error.

```
       01  ERROR-COUNT-TABLE.
           02  EACH-ROW                     OCCURS 4 TIMES.
               03  ERROR-COUNT              PICTURE IS 9(06).
```

Figure 17.12
Definition of a table for frequency counts

Further, assume PAYROLL-IDENTIFICATION is the first field in each time card. This field must contain a numeric value or it is considered to be in error. This known, the edit logic might very well be constructed as shown in figure 17.13. This newest ADD instruction would be included in each field examination. Of course, the subscript would be different for each field. The second field in TIME-CARD-INPUT would have a subscript of two, the third three, and so on. Also, the MAIN-CONTROL SECTION would have to be modified to report those statistics.

Another problem still remains in this latest application of tables. The table ERROR-COUNT-TABLE contains a multi-valued variable ERROR-COUNT that has no initial value. As a result, the first ADD instruction using ERROR-COUNT causes the abnormal termination of the edit program. This problem can be solved by redefining ERROR-COUNT-TABLE as shown in figure 17.14. Figure 17.14 shows that we have four FILLER variables, six digits in length and initialized to zero, occupying the same storage locations as the four rows where each row is six digits long.

An alternative might involve the segments from a COBOL program shown in figure 17.15. This alternative requires a table to be assigned its initial values while the program is executing rather than at the time the program is compiled by the COBOL compiler. While this method works quite well, it has the disadvantage of requiring additional executable instructions, which increases program size and execution time.

Figure 17.13
Incrementing frequency counts in tables

```
           IF PAYROLL-IDENTIFICATION
                   OF TIME-CARD-INPUT NOT NUMERIC
               THEN
                   ADD 1 TO ERRORS-FOUND OF SUBTOTALS
                   ADD 1 TO CURRENT-LINE
                   MOVE "YES" TO TRANSACTION-STATUS
                   ADD 1 TO ERROR-COUNT (1)
                   WRITE PRINT-RECORD
                       FROM
                           ERROR-MESSAGE-4
                       AFTER ADVANCING
                           1 LINES.
```

Figure 17.14
Initializing frequency count tables

```
       01  ERROR-COUNT-TABLE.
           02  DATA-PORTION.
               03  FILLER          PICTURE IS 9(06) VALUE ZERO.
               03  FILLER          PICTURE IS 9(06) VALUE ZERO.
               03  FILLER          PICTURE IS 9(06) VALUE ZERO.
               03  FILLER          PICTURE IS 9(06) VALUE ZERO.
           02  EACH-ROW            REDEFINES DATA-PORTION
                                   OCCURS 4 TIMES.
               03  ERROR-COUNT     PICTURE IS 9(06).
```

Figure 17.15
An alternative in frequency table initialization

```
       01  ERROR-COUNT-TABLE.
           02  EACH-ROW            OCCURS 4 TIMES.
               03  ERROR-COUNT     PICTURE IS 9(06).
             .
             .
             .
           MOVE ZERO TO ERROR-COUNT (1).
           MOVE ZERO TO ERROR-COUNT (2).
           MOVE ZERO TO ERROR-COUNT (3).
           MOVE ZERO TO ERROR-COUNT (4).
```

Column and Row Orientation

The HOURLY-RATE-TABLE introduced in figures 17.3 and 17.4 consists of eleven rows and two columns. The multi-valued variable PAYROLL-CATEGORY might be thought of as column 1, while HOURLY-RATE is column 2. Both variables were defined as 03-level items subordinate to the OCCURS clause in an 02-level item. This form of table definition is known as *row major order.*

When tables are defined in row major order, they are arranged in computer storage row by row. That is, all of the columns in row 1 appear one after the other, followed by all of the columns in row 2, and so on. (See figure 17.16.)

Row major order requires that tables be assigned their initial values on a row-by-row basis. Therefore, the table first defined in figure 17.4 has a nonnumeric literal for DATA-PORTION that is a series of row values one after the other. For instance, DL-1310 is followed by DL-2345, and so on. The first row value, DL-1310, represents the contents of both columns in the first row of the HOURLY-RATE-TABLE. DL-1 is the PAYROLL-CATEGORY while 310 is the HOURLY-RATE.

The HOURLY-RATE-TABLE can also be defined using a *column major order.* As figure 17.17 illustrates, HOURLY-RATE-TABLE, without its initial values, is defined somewhat differently.

The table definition in figure 17.17 is different from earlier attempts to define HOURLY-RATE-TABLE because each of the multi-valued variables contains its own OCCURS clause. The table still requires a total of seventy-seven storage positions. However, the allocation of those positions to the variables PAYROLL-CATEGORY and HOURLY-RATE is quite different. Rather than having one table with eleven rows each with two columns, we now have two tables each with eleven rows and one column. In graphic terms, as figure 17.18 depicts, all of the values for PAYROLL-CATEGORY are grouped together and followed by all of the values for HOURLY-RATE.

Row Major Order
A process for establishing tables in primary memory so that all of the columns in row 1 come first, followed by all of the columns in row 2, followed by all of the columns in row 3, and so on.

Column Major Order
A process for establishing tables in primary memory so that all of the rows in column 1 come first, followed by all of the rows in column 2, followed by all of the rows in column 3, and so on.

ROW 1	ROW 1	ROW 2	ROW 2	ROW 11
COL 1	COL 2	COL 1	COL 2	COL 2

Figure 17.16
Storage map for row major order tables

```
       01  HOURLY-RATE-TABLE.
           02  TABLE-PORTION.
               03  PAYROLL-CATEGORY          OCCURS 11 TIMES
                                             PICTURE IS X(04).
               03  HOURLY-RATE               OCCURS 11 TIMES
                                             PICTURE IS 9V99.
```

Figure 17.17
HOURLY RATE TABLE using column major order

DL-1	DL-2	DL-3	DL-4	SL-1	SL-2	....	SO-2

310	345	375	435	510	626	....	785

Figure 17.18
Storage map for column major order tables

The processing logic depicted in figures 17.5 through 17.9 need not be changed if HOURLY-RATE-TABLE is defined by using column major order rather than row major order. This is because the COBOL compiler produces the appropriate executable coding based upon the table definition.

One more point requires discussion. Note the alternative definitions in figure 17.19. Does it make any difference how the data in each table is actually placed in storage? The answer is no. All alternatives will yield identical storage maps. The maps, if produced, would look like those shown in figure 17.19.

Figure 17.19
Alternatives in column major order definition

Alternative one

```
       01  HOURLY-RATE-TABLE.
           02 DATA-PORTION               PICTURE IS X(77) VALUE
           "DL-1DL-2DL-3DL-4SL-1SL-2OW-1OW-2OW-3SO-1SO-23103453754355106
           "26366488525655785".
           02 TABLE-PORTION              REDEFINES DATA-PORTION.
               03 PAYROLL-CATEGORY       OCCURS 11 TIMES
                                         PICTURE IS X(04).
               03 HOURLY-RATE            OCCURS 11 TIMES
                                         PICTURE IS 9V99.
```

Alternative two

```
       01  HOURLY-RATE-TABLE-1.
           02 DATA-PORTION                PICTURE IS X(44) VALUE
           "DL-1DL-2DL-3DL-4SL-1SL-2OW-1OW-2OW-3SO-1SO-2".
           02 PAYROLL-CATEGORY            REDEFINES DATA-PORTION
                                          OCCURS 11 TIMES
                                          PICTURE IS X(04).
       01  HOURLY-RATE-TABLE-2.
           02 DATA-PORTION                PICTURE IS X(33) VALUE
           "310345375435510626366488525655785".
           02 HOURLY-RATE                 REDEFINES DATA-PORTION
                                          OCCURS 11 TIMES
                                          PICTURE IS 9V99.
```

Alternative three

```
       01  HOURLY-RATE-TABLE.
           02 PART-1.
               03 DATA-PORTION            PICTURE IS X(44) VALUE
               "DL-1DL-2DL-3DL-4SL-1SL-2OW-1OW-2OW-3SO-1SO-2".
               03 PAYROLL-CATEGORY        REDEFINES DATA-PORTION
                                          OCCURS 11 TIMES
                                          PICTURE IS X(04).
           02 PART-2
               03 DATA-PORTION            PICTURE IS X(33) VALUE
               "310345375435510626366488525655785".
               03 HOURLY-RATE             REDEFINES DATA-PORTION
                                          OCCURS 11 TIMES
                                          PICTURE IS 9V99.
```

Alternative four

```
       01  HOURLY-RATE-TABLE.
           02 DATA-PORTION-1             PICTURE IS X(44) VALUE
           "DL-1DL-2DL-3DL-4SL-1SL-2OW-1OW-2OW-3SO-1SO-2".
           02 PAYROLL-CATEGORY           REDEFINES DATA-PORTION-1
                                         OCCURS 11 TIMES
                                         PICTURE IS X(04).
           02 DATA-PORTION-2             PICTURE IS X(33) VALUE
           "310345375435510626366488525655785".
           02 HOURLY-RATE                REDEFINES DATA-PORTION-2
                                         OCCURS 11 TIMES
                                         PICTURE IS 9V99.
```

Summary

Multi-valued variables play a significant role in the design and construction of computer programs. These variables can simplify the programming task by allowing the programmer to construct iterative logic. This iterative logic provides access to each possible value that a multi-valued variable can possibly hold. The alternative requires the construction of very large, single statements. PAYROLL-CATEGORY in the edit program serves as an excellent example.

PAYROLL-CATEGORY from TIME-CARD-INPUT can contain one of many possible values. The edit program can examine PAYROLL-CATEGORY in a single IF statement containing many conditions connected by an AND. If the values that PAYROLL-CATEGORY can hold are relatively few in number, the single IF statement might be quite reasonable. If, on the other hand, PAYROLL-CATEGORY contains as many as 200 different values, a table of those values and the necessary iterative processing logic would be the logical alternative.

Table definitions are much like record descriptions. In fact, a record description might be viewed as a one-row table. Tables are created out of record descriptions by the addition of an entry with an OCCURS clause. The OCCURS clause specifies the number of rows a table is to contain.

Tables, like record definitions, can be assigned initial values. Unlike record definitions, however, these initial values cannot be assigned by using the VALUE clause after each variable definition in the table. Instead, the programmer can assign initial values in one of two possible ways.

The first method of table initialization requires the use of storage/variable redefinition. Here one or more variables with the appropriate initial values are defined. This value, or set of values, is then redefined by the rows of the table which cause the values and the rows to occupy the same locations of computer storage.

The second method of table initialization involves adding instructions to a program's PROCEDURE DIVISION. This permits the programmer to assign initial values row by row. Unfortunately, this method adds additional executable instuctions to COBOL programs, thereby increasing the size of the program and execution time requirements.

The program size and execution time should be minimized whenever possible. In this regard, table manipulation with subscripts defined as externally formatted variables cause increases in these two areas. For this reason, the SEARCH verb and its associated index variable should be used. Index variables access tables by row addresses, which do not require the conversion that DISPLAY, COMPUTATIONAL, and COMPUTATIONAL-3 variables need.

Additionally, the SEARCH verb accomplishes the iterative examination of table entries in fewer statements. The construction of iterative logic using MOVE, ADD, IF, and GO statements can lead to a specification error in program logic that may not be easily found. While PERFORM statements help to reduce the problems of controlling iterative functions, the actual operation of the UNTIL clause can cause the last row of the table to be ignored during the search. This possible, and difficult to catch, error often prompts many to consider the SEARCH verb as the preferred table-searching mechanism.

The SEARCH verb requires table definitions to be modified to include the phrase *INDEXED BY*. The variable name supplied as the index will be automatically defined by the COBOL compiler as an index variable. This variable must be initialized with the address of the row in the table at which the search is to begin. This initialization can only be completed by the use of the SET instruction. An attempt to initialize an index variable by any other means will not be accepted by the COBOL compiler.

The SEARCH verb automatically checks each row of the table for the value desired. If the value is not found, control passes to the COBOL statements that are a part of the AT END segment. The WHEN segment contains two types of statements. The first is the search criteria stating when the search is considered satisfied. The second is what to do when the search criteria have been satisfied.

Finally, tables in COBOL programs can be defined so their contents appear allocated in storage in either column major order or row major order. Neither method, however, has any impact upon the construction of the processing logic in the program's PROCEDURE DIVISION. Use of either alternative will result in the COBOL compiler generating the appropriate storage-mapping function—the executable code that permits access to each data item within a table.

Notes

1. These alternatives in internal representation of numeric data apply to the IBM 360, 370, 303X, 43X1, and 308X computers only. Other computers may feature other alternatives. Consult the vendor reference literature for the COBOL compiler in use at your computing installation.

Questions for Review

1. Why are tables useful in COBOL programs?
2. Under what circumstances should tables, rather than condition variables, be used? In what aspects are they similar? Dissimilar?
3. Why are the SEARCH and PERFORM verbs preferred for table-processing applications, while combinations of IFs, GO TOs, and ADD are not?
4. "Programs searching large tables must use subscript variables that minimize the use of computer resources in the search process." Is this correct? Explain.
5. Which of the two verbs, SEARCH or PERFORM, are best suited for searching tables?

Problems

1. What, if anything, is wrong with the following table declaration?

```
01  STATE-ABBREV-TABLE.
    02 EACH-ENTRY           OCCURRS 50 TIME
       03 STATE             PICTURE IS X(03).
```

2. Revise your corrected version of the table declaration in exercise 1 to include the table's values. Can table initialization be accomplished in some other manner? Is any one method preferred?
3. Assume that

```
01  ROW-POINTER      PICTURE IS 9(01).
```

   is to serve as the subscript variable for the table defined in exercise 2. Will there be a problem in its use? Explain.
4. Assume that STATE-ABBREV-TABLE is to be used in a program charged with examining time cards for a payroll system. Each time card contains the employee's state of residence. Write the COBOL segments to search the table for a matching state abbreviation, using (a) GO TOs and IFs,
(b) PERFORM, and (c) SEARCH

5. Examine the following table declaration. Will it work as the author intended? Why or why not?

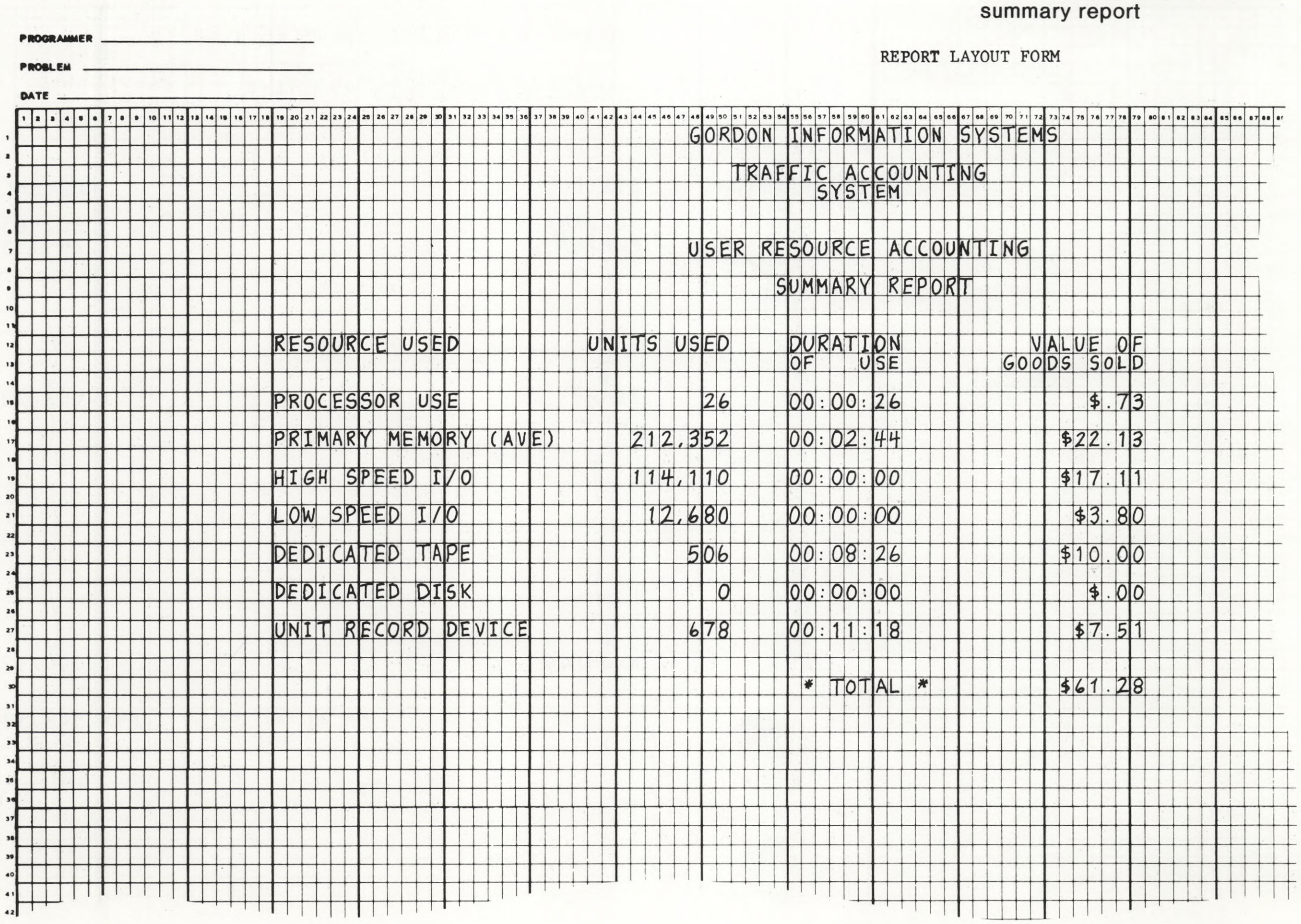
```
01   HOURLY-RATE-TABLE.
     02 DATA-PORTION              PICTURE IS X(77) VALUE
     "DL-1310DL-2345DL-3375DL-4435SL-1510SL-26260W-13660W-24880W-3
-    "525SO-1655SC-2785".
     02 TABLE-PORTION             REDEFINES DATA-PORTION.
        03 PAYROLL-CATEGORY       OCCURS 11 TIMES
                                  PICTURES IS X(04).
        03 HOURLY-RATE            OCCURS 11 TIMES
                                  PICTURE IS 9V99.
```

6. Why is it that figure 17.19 sometimes requires the use of DATA-PORTION-1 and DATA-PORTION-2, while at other times DATA-PORTION is acceptable?

Programming Assignments

1. Gordon Information Services, the successors to Thomas Information Systems, wishes to amend the program/report originally developed in chapter 11. In addition to the report specified in figure 11.26, the new owners also desire a supplemental summary report like that shown in figure 17.20. This new report is to be produced by the same program from chapter 11. However, the summary report is to be produced by a file separate from that used to generate the original report. This will require a new SELECT, FD, etc.

Figure 17.20
Gordon Information Systems summary report



2. Johnston Mailing Service provides automatic mailing services for bulk mailings. In addition to using the mailing lists of their clients, Johnston also maintains its own mailing list which it rents to clients for a set fee. One measure of worth is an analysis of how well Johnston's mailing list covers the continental United States. Consequently, Johnston's has developed a computer program that provides a summary of how many of its destinations are to be found within each state. The report produced by this program can be found in figure 17.21. The input to this program is, of course, the mailing list whose records adhere to the following format:

Record Position	*Data Type*	*Variable*
01–02	AN	Two character state abbreviation
03–11	N	Zipcode
12–30	AN	Intended recipient
31–45	AN	Mail carrier route address
46–60	AN	City
61–75	AN	County
76–80	AN	Demographic characteristics

You are to produce the program that will generate this report from the data listed.

Figure 17.21
Johnston Mailing Services state report

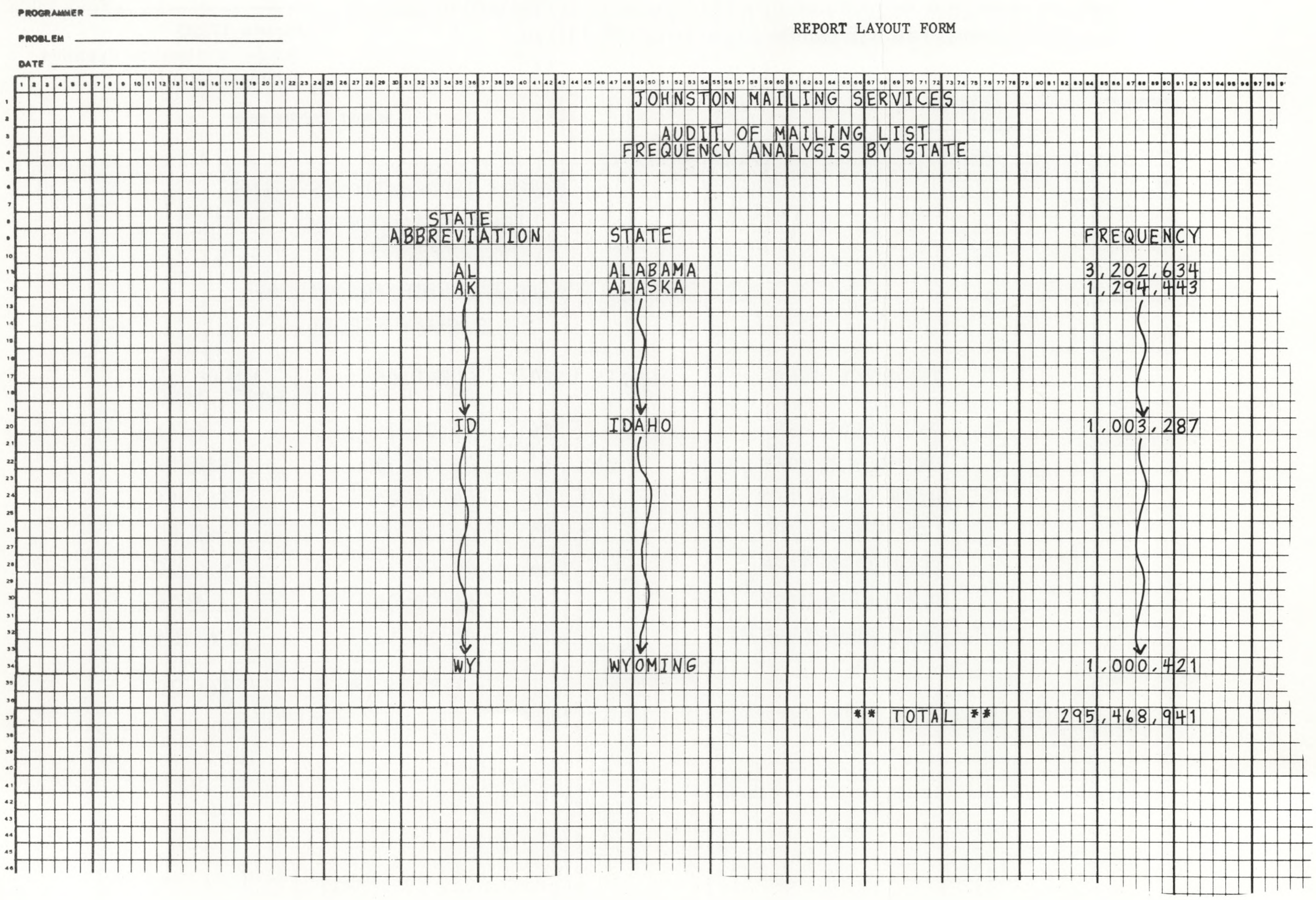


Table Definition and Processing
Part 2

18

Chapter Outline

Tables are an indispensable data structure in any programming language. They can also be quite confusing to the beginning programmer. It is not surprising, therefore, that many beginners can be coaxed into learning how to define the process only the most very basic of multi-value variables—seldom anything more. Unfortunately, even the beginning programmer needs to know more about this particular type of data structure. Without additional knowledge in definition and use, the beginner will design programs that are difficult to handle, causing distress during their initial development and even more undesirable outcomes for other programmers during the inevitable maintenance portion of the software life cycle. For this reason, the material concerning table definition and use has been partitioned into two separate chapters.

The first chapter introduced only the most fundamental concepts associated with tables. From this material you should know how to define, assign initial values to, and access the elements within simple table structures. In addition, you should know how to define and manipulate subscript variables that are the least efficient to the most efficient, from a machine standpoint. Hopefully, programming assignments accompanying the initial chapter on this topic will prove that tables are not as difficult as they seem. This chapter, however, attempts to go one step farther.

This chapter introduces three new concepts in table definition and use. The first involves improving the efficiency of table search operations, the second with synchronizing access to two-distinct tables, and the third with tables featuring two or more dimensions. Each of these concepts requires a thorough understanding of the basics introduced in the chapter 17. Consequently, you are encouraged to review the earlier materials on this topic before continuing.

Binary Searches

Most of the tables described thus far have been relatively short. That is, these tables have featured a small number of rows, or entries, as we sometimes refer to them. Thus, the report program logic described in figure 18.1 requires a search of a table until a match has been located for each input record. The results of processing are then published.

A table containing a small number of rows requires very little machine time to search it from beginning to end. Tables with larger numbers of rows require larger amounts of computing resources to search them. Perhaps the following example will better illustrate this point.

Assume a report program is to reformat and print each record in an input file. Report programs also perform other activities such as page monitoring, control break monitoring, accumulating subtotals, and so on. These will be ignored for the present. What is of interest at the moment is the act of reformatting the data in each input record to produce an easy-to-read report.

Further, assume that each input record contains a two-digit alphabetic abbreviation of a state in the union. Thus, VA represents Virginia. Rather than print the abbreviation, a fuller representation should appear on the report. For this reason, the report program will contain a table like that partially shown in figure 18.2. The major question here is quite simple. On the average, how many of the fifty rows in the table shown in figure 18.2 must be searched to find a match?[1]

The answer is twenty-five. On the average, it is likely we will have to search twenty-five of the fifty rows for each input record, before a match is found between the two-digit abbreviations in the input records and the table.

Today's processing units execute many thousands of instructions per second. Indeed, many machines execute instructions at even faster speeds than this. As a result, it may seem needless to worry about how many comparison operations are necessary before items can be located in tables. No rule states precisely the number of rows a table can possess before a programmer should be concerned about lengthy search procedures. However, to lessen the time required to search a table, a solution does exist.

```
Print and Read Process

        Process data in record
          .
          .
            (search a table)
          .
          .
        Print (publish image of present
               record to be processed)
        Read next record to process
```

Figure 18.1
Practical use of table processing

ABBREVIATION	Fuller representation
AL	ALABAMA
.	.
.	.
.	.
MO	MISSOURI
.	.
.	.
.	.
WY	WYOMING

Figure 18.2
State abbreviation table

Ordered Entries

The only table search strategy used thus far has been serial in nature. The examination process requires that the first row of a table be inspected for the item desired. If no match is found in that row, the examination continues in the next, or second, row. If still no match is found, the examination proceeds to the third row, and so on through each row in the table. This particular strategy assumes random or unordered data in the table.

The state abbreviations shown in figure 18.2 are arranged alphabetically from the lowest to the highest, using the usual collating sequence of A before B before C. We normally assume, in report programs, that the data being read and processed from the input file has already passed an edit procedure. In the example involving state abbreviations, this means each of the two-digit abbreviations in the input file is legitimate. Suppose, for the sake of argument, that this cannot be guaranteed. That is, one record in the input file contains a state abbreviation of DF, and no state has the abbreviation DF.

Since the abbreviation-name pairs are ordered alphabetically, the state of Delaware (DE) is immediately followed in the table by Florida (FL). In our collating sequence, DF is greater than DE but smaller than FL. Thus when DE and DF are compared, our program logic directs the search to proceed to the next row in the table. Here, however, DF is smaller than FL. Consequently, the search need not continue, and since DF does not match FL, no match is found in the entire table. No other entry in the table has the first character D. Florida marks the start of F or greater.

We can use this knowledge of the ordering of table entries to construct a nonserial search strategy to minimize the average number of comparison operations required to find a match between the abbreviations in the input records and the table. This nonserial search strategy requires partitioning the table into a series of smaller tables.

Partitioning the Table

The first twenty-five states in an ordered state abbreviation table encompass Alabama (AL) through Missouri (MO). The second twenty-five states encompass Montana (MT) through Wyoming (WY). This perspective of the table can be seen in figure 18.3. As the newest version of the abbreviation-name table discloses, if an abbreviation DF exists, it must be in the first half of the ordered table.

Clearly, DF falls within the range AL through MO. As a result, if we go to the midpoint of our table we can determine in which half of the table a match will be found, if one exists at all. This allows us, with the benefit of only one comparison, to eliminate twenty-five rows that need not be examined. If the table is not divided any further, on the average only thirteen rows need ever be examined to find a match between abbreviation names. Our intent, however, is to divide the table even further.

What if the half of the table in which a match must be found were itself divided in half? This is shown in figure 18.4. The term half is used somewhat liberally here as one grouping contains twelve entries while the second contains thirteen. Does this provide the opportunity to further improve the search process? The answer can only be yes.

As an inspection of the table in figure 18.4 reveals, if a state has the abbreviation DF, the entry for that state must be in the first quarter of the table. By partitioning the table into quarters we have, with just two comparison operations, eliminated thirty-eight states from the searching process. If a match is to be found, at most twelve state entries must be examined. On the average, however, once the proper quadrant of the table is found, only six additional comparison operations would ever be required.

Should the partitioning of the table into successively smaller portions be terminated after the creation of quadrants? If yes, we would still have made a considerable improvement over the serial search strategy described earlier. By partitioning the table into quadrants, which requires only two comparison operations, we would on the average be required to serially search seven entries in order to locate a possible match. However, this can be further improved.

If the table can be divided into quadrants, why not into eighths? That is, if we learn that a possible match can be isolated to a particular quadrant, why not partition each quadrant into approximate halves? The match desired could immediately be determined to be in one of the newest halves (i.e., actually eighths), while not in the others. This division into eighths requires only three comparison operations and leaves a segment of four entries to be serially searched. On the average, only two of the four entries will require examination.

By successively dividing the state table into halves, where each partitioning requires a single comparison operation, we can greatly improve the performance of the computer in searching table entries. This particular method of directed search is known as a *binary search*. A binary search strategy requires that the partitioning process continue until nothing is left to split into halves. That is, the last division operation will result in halves that each contain one row.

Binary Search
A directed search that successively partitions a monotonically ordered table into smaller halves. By focusing the search upon the segment of the table that is likely to contain the item desired, if it is in the table at all, a significant part of the table is not examined.

Writing a section of COBOL coding to perform a binary search is more than a trivial undertaking. The designers of the COBOL language have provided a way to automatically specify a binary search if one is desired. Examine the COBOL definition for a possible state abbreviation-name table shown in figure 18.5.

The COBOL definition shown in figure 18.5 is similar to the one introduced in an earlier chapter. The definition consists of two parts. The first part specifies the data that the table is to contain. The data is fifty consecutive strings, each of which is sixteen positions in length. The first two of the sixteen positions provides the abbreviation for a particular state. The remaining fourteen positions in each string supply the complete spelling that corresponds to each abbreviation.

AL	ALABAMA	MT	MONTANA
AK	ALASKA	NB	NEBRASKA
.	.	.	.
.	.	.	.
.	.	.	.
MI	MISSISSIPPI	WV	WEST VIRGINIA
MO	MISSOURI	WY	WYOMING

Figure 18.3
State table divided into halves

AL	ALABAMA	IL	ILLINOIS
AK	ALASKA	IN	INDIANA
.	.	.	.
.	.	.	.
.	.	.	.
ID	IDAHO	MI	MISSISSIPPI
		MO	MISSOURI

Figure 18.4
First two quarters of state table

```
           SET STATE-INDEX TO 1.
           SEARCH EACH-STATE-ENTRY
               WHEN
                   STATE-ABBREVIATION
                       OF ABBREVIATION-NAME-TABLE (STATE-INDEX) =
                   STATE-ABBREVIATION
                       OF INPUT-RECORD-DESCRIPTION
                           MOVE STATE-NAME
                               OF ABBREVIATION-NAME-TABLE (STATE-INDEX)
                                           TO
                                STATE-NAME
                               OF OUTPUT-RECORD-DESCRIPTION.
```

Figure 18.5
COBOL state abbreviation-name table

The second part of the definition constitutes a template to be overlaid upon the data. This template describes that each of the fifty strings has a common structure. The first two positions, or state abbreviation, is known symbolically as STATE-ABBREVIATION. The remaining fourteen positions, or state name, is accessed by the variable STATE-NAME. Both STATE-ABBREVIATION and STATE-NAME are multi-valued variables. Consequently, the subscript, STATE-INDEX, must be set to indicate which of the fifty possible values the program wishes to access.

The examination logic shown in figure 18.6 provides a serial search that, on the average, will require an inspection of twenty-five of the fifty rows in order to locate a match. That assumes that a match does, in fact, exist. On the other hand, when it is possible that no match exists, all fifty rows will have to be examined to determine that fact. Consequently, edit programs relying upon strict serial searches of table structures will have to examine, on the average, far more than half of all rows in the table. Report programs processing error-free data, however, need only examine, on the average, half of all table entries.

Figure 18.6
Serial search assumes existence of a match

```
.
.
(reformat first portion of input record for purposes of
 printing)
.
.
SET STATE-INDEX TO 1.
SEARCH EACH-STATE-ENTRY
    WHEN
        STATE-ABBREVIATION
            OF ABBREVIATION-NAME-TABLE (STATE-INDEX) =
        STATE-ABBREVIATION
            OF INPUT-RECORD-DESCRIPTION
                MOVE STATE-NAME
                    OF ABBREVIATION-NAME-TABLE (STATE-INDEX)
                                 TO
                     STATE-NAME
                    OF OUTPUT-RECORD-DESCRIPTION.
.
.
(reformat remainder of input record for purposes of printing)
.
.
```

Statement Format

```
8/12
↓
levelnumber  repeateditem  OCCURS n TIMES
                           {ASCENDING }
                           {DESCENDING} KEY IS variable

                           INDEXED BY indexname.
```

If a binary search is desired, the table definition must be altered. Specifically, the COBOL compiler must be informed of two things. It must be given the two multivalued variables (STATE-ABBREVIATION or STATE-NAME) in the table that will be used for the binary search. It must also be told whether the values within the key variable are arranged in ascending or descending order according to the collating sequence. A slightly modified COBOL definition that accomplishes both tasks is shown in figure 18.7.

The phrase ASCENDING KEY IS informs the COBOL compiler that, in this case, the binary search is to be performed using the ascendingly, ordered values in the variable STATE-ABBREVIATION to guide the partitioning operations that characterize this nonserial search strategy.

The SEARCH verb requires only slight modification to complete the request for a binary, as opposed to a serial, search. This is accomplished by omitting the SET verb and following the word SEARCH with the word ALL. The SET verb is unnecessary, because the binary search logic will automatically determine where in the table the partitioning process is to begin.

Parallel Table Access

Frequently, a programmer may choose to define table structures so their contents are homogeneous. The table definitions shown in figures 18.6 and 18.7 are not homogeneous. The state abbreviation table contains two distinct and different types of data—abbreviations and fully spelled names. A homogeneous version of the abbreviation table can be found in figure 18.8.

Page 1 of 2

```
       01  ABBREVIATION-NAME-TABLE.
           02 DATA-PORTION.
              03 ALABAMA                PICTURE IS X(16) VALUE
              "ALALABAMA".
              03 ALASKA                 PICTURE IS X(16) VALUE
              "AKALASKA".
              .
              .
              .
              03 WEST-VIRGINIA          PICTURE IS X(16) VALUE
              "WVWEST VIRGINIA".
              03 WYOMING                PICTURE IS X(16) VALUE
              "WYWYOMING".
           02 EACH-STATE-ENTRY          REDEFINES DATA-PORTION
                                        ASCENDING KEY IS
                                             STATE-ABBREVIATION
                                        OCCURS 50 TIMES
                                        INDEXED BY STATE-INDEX.
              03 STATE-ABBREVIATION     PICTURE IS X(02).
              03 STATE-NAME             PICTURE IS X(14).
           SEARCH ALL EACH-STATE-ENTRY
               WHEN
                   STATE-ABBREVIATION
                       OF ABBREVIATION-NAME-TABLE (STATE-INDEX) =
```

Page 2 of 2

```
                   STATE-ABBREVIATION
                       OF INPUT-RECORD-DESCRIPTION
                           MOVE STATE-NAME
                               OF ABBREVIATION-NAME-TABLE (STATE-INDEX)
                                    TO
                                 STATE-NAME
                               OF OUTPUT-RECORD-DESCRIPTION.
```

Figure 18.7
COBOL definition and request for binary search

Page 1 of 2

```
       01  STATE-ABBREVIATION-TABLE.
           02 DATA-PORTION              PICTURE IS X(100) VALUE
              "ALASAKAZCACO..........
      -       "NCND............................WVWY".
           02 EACH-ABBREVIATION         REDEFINES DATA-PORTION
                                        OCCURS 50 TIMES
                                        INDEXED BY
                                            ABBREVIATION-INDEX.
              03 STATE-ABBREVIATION     PICTURE IS X(02).

       01  STATE-NAME-TABLE.
           02 DATA-PORTION.
              03 ALABAMA                PICTURE IS X(14) VALUE
              "ALABAMA".
              03 ALASKA                 PICTURE IS X(14) VALUE
              "ALASKA".
              .
              .
              03 WEST-VIRGINIA          PICTURE IS X(14) VALUE
              "WEST VIRGINIA".
              03 WYOMING                PICTURE IS X(14) VALUE
              "WYOMING".
           02 EACH-NAME                 REDEFINES DATA-PORTION
```

Page 2 of 2

```
                                        OCCURS 50 TIMES
                                        INDEXED BY
                                            NAME-INDEX.
              03 STATE-NAME             PICTURE IS X(14).
```

Figure 18.8
Two separate tables for state abbreviations and names

Figure 18.9
Unpredictable MOVE due to NAME-INDEX value

```
           SET ABBREVIATION-INDEX TO 1.
           SEARCH EACH-ABBREVIATION
               WHEN
                   STATE-ABBREVIATION
                       OF STATE-ABBREVIATION-TABLE
                               (ABBREVIATION-INDEX) =
                   STATE-ABBREVIATION
                       OF INPUT-RECORD-DESCRIPTION
                           MOVE STATE-NAME
                               OF STATE-ABBREVIATION-TABLE
                                   (ABBREVIATION-INDEX) TO
                           STATE-NAME
                               OF OUTPUT-RECORD-DESCRIPTION.
```

Figure 18.10
Corrected manipulation of NAME-INDEX

```
           SET ABBREVIATION-INDEX TO 1.
           SEARCH EACH-ABBREVIATION
               VARYING
                   NAME-INDEX
               WHEN
                   STATE-ABBREVIATION
                     OF STATE-ABBREVIATION-TABLE (ABBREVIATION-INDEX) =
                   STATE-ABBREVIATION
                     OF INPUT-RECORD-DESCRIPTION
                           MOVE STATE-NAME
                               OF STATE-NAME-TABLE (NAME-INDEX) TO
                               STATE-NAME
                               OF OUTPUT-RECORD-DESCRIPTION.
```

Now two tables exist where previously only one had been. Each of these newest tables is homogeneous. Unfortunately, each table is indexed by its own subscript. The table of abbreviations is INDEXED BY ABBREVIATION-INDEX, while the table of fully spelled names is INDEXED BY NAME-INDEX. This poses a problem, but the solution is really quite simple.

If ABBREVIATION-INDEX is manipulated to gain access to each of the rows in the table of abbreviations, what happens to NAME-INDEX, which belongs to an entirely different table? The answer is that nothing is done to NAME-INDEX unless you direct the SEARCH logic to do otherwise. For example, examine the SEARCH logic that is found in figure 18.9. Notice that only ABBREVIATION-INDEX is being SET. Since no action has been taken to also set and manipulate NAME-INDEX, the MOVE instruction will produce unpredictable results. The results will be unpredictable because we have no way of knowing what value (i.e., row) NAME-INDEX contains.

Neither the SEARCH verb nor the SET verb mention the value or treatment of NAME-INDEX. Consequently, nothing will be done with NAME-INDEX during the SEARCH, and as a result, the MOVE instruction is destined to provide an undesirable outcome. A slight alteration, however, can overcome this problem.

In the search logic shown in figure 18.10, the SEARCH verb guarantees that, at any given point in time, NAME-INDEX will point to the same row number as does ABBREVIATION-INDEX. Thus, if ABBREVIATION-INDEX is subscripting row 1 in the abbreviation table, then NAME-INDEX will also point to row 1 in the table of state names. The MOVE instruction will now work as anticipated.

Step level	0	1	2	3	4	5
1	8,500	8,750	9,000	9,250	9,500	9,750
2	10,500	10,750	11,000	11,250	11,500	11,750
3	12,500	12,750	13,000	13,250	13,500	13,750
4	14,500	14,750	15,000	15,250	15,500	15,750
5	17,000	17,500	18,000	18,500	19,000	19,500
6	22,500	23,000	23,500	24,000	24,500	25,000
7	27,000	27,500	28,000	28,500	29,000	29,500

Figure 18.11
Salary scale expressed in matrix form

```
       01  SALARY-MATRIX.
           02 DATA-PORTION.
              03 FIRST-LEVEL           PICTURE IS X(36) VALUE
              "085000875009000092500950009750".
              03 SECOND-LEVEL          PICTURE IS X(36) VALUE
              "105001075011000112501150011750".
              03 THIRD-LEVEL           PICTURE IS X(36) VALUE
              "125001275013000132501350013750".
              03 FOURTH-LEVEL          PICTURE IS X(36) VALUE
              "145001475015000152501550015750".
              03 FIFTH-LEVEL           PICTURE IS X(36) VALUE
              "170001750018000185001900019500".
              03 SIXTH-LEVEL           PICTURE IS X(36) VALUE
              "225002300023500240002450025000".
              03 SEVENTH-LEVEL         PICTURE IS X(36) VALUE
              "270002750028000285002900029500".
           02 EACH-SALARY-LEVEL        REDEFINES DATA-PORTION
                                       OCCURS 07 TIMES
                                       PICTURE IS X(36).
```

Figure 18.12
Incomplete COBOL definition of a salary matrix

Tables Requiring Two Subscripts

Until now, all of the tables defined and processed have required the use of only one subscript variable. Our tables have contained multiple rows, and the subscript informed the computer which row we desired to access. Tables can be defined so that we must also specify which column we wish to access as well.

Many organizations feature a salary schedule that can be expressed in matrix form as shown in figure 18.11. The rows in such a salary matrix represent base salary levels that are paid to an organization's employees. The columns may represent yearly steps within each salary level. Step increases may be awarded to employees on the anniversary of their employment with the organization. As an inspection of the matrix reveals, step increases are smaller than level increases. Level increases may be awarded only when an employee has been promoted within the organization or is deserving of a larger base salary.

When salary matrices are used, an individual's salary can be determined only if both level and step number are known. In terms of table definition and processing, to determine salary we must know the row subscript value and the column subscript value.

The salary matrix shown in figure 18.11 can be seen as a COBOL table definition in figure 18.12. As this latest definition illustrates, tables with rows and columns are defined one row at a time. First, all of the values to be found in the table's initial row can be found, followed by those of the second row, the third row, and so on.

Figure 18.13
Processing an incompletely defined salary matrix

```
           MOVE 3 TO SALARY-INDEX.
           DISPLAY EACH-SALARY-LEVEL.
```

Figure 18.14
Completed COBOL definition of a salary matrix

```
       01  SALARY-MATRIX.
           02 DATA-PORTION.
              03 FIRST-LEVEL              PICTURE IS X(36) VALUE
              "085000875009000092500950009750".
              03 SECOND-LEVEL             PICTURE IS X(36) VALUE
              "105001075011000112501150011750".
              03 THIRD-LEVEL              PICTURE IS X(36) VALUE
              "125001275013000132501350013750".
              03 FOURTH-LEVEL             PICTURE IS X(36) VALUE
              "145001475015000152501550015750".
              03 FIFTH-LEVEL              PICTURE IS X(36) VALUE
              "170001750018000185001900019500".
              03 SIXTH-LEVEL              PICTURE IS X(36) VALUE
              "225002300023500240002450025000".
              03 SEVENTH-LEVEL            PICTURE IS X(36) VALUE
              "270002750028000285002900029500".
           02 EACH-SALARY-LEVEL           REDEFINES DATA-PORTION
                                          OCCURS 07 TIMES.
              03 EACH-SALARY-STEP         OCCURS 06 TIMES.
                 04 SALARY                PICTURE IS 9(05).
```

The COBOL table definition shown in figure 18.12 is almost complete. Like all other definitions examined thus far, this table requires the use of only one subscript. Assume the subscript is a numeric variable defined in the WORKING-STORAGE SECTION as SALARY-INDEX. What results would the section of COBOL coding shown in figure 18.13 produce?

The two statements in figure 18.13 would print the contents of the entire third row of the salary matrix. That is, the six consecutive salaries starting with $12,500 would print without any intervening blanks or editing characters. To gain individual access to just one of those six salaries in the third row requires the row to be subdivided into columns using yet another OCCURS clause. After all, salaries occur six times in each row, as shown in figure 18.14.

If an employee's salary level and step are known, the employee's salary can be extracted from the salary matrix. Obviously, these two data items would have to be a part of each employee's payroll authorization record in the payroll master file. If one or both of these items are unknown for any employee, it is impossible to access that employee's salary from the matrix.

Assume that we are to construct a report program that reformats the data from each employee's payroll authorization record from the payroll master file. The sequence required to convert an employee's salary level and step data into a yearly salary can be seen in figure 18.15. Note that because SALARY is subordinate to two OCCURS clauses, two subscripts are required. An attempt to use only one subscript does not provide sufficient information about the salary desired.

```
       01  PAYROLL-AUTHORIZATION-RECORD.
           02 EMPLOYEE-NAME                       PICTURE IS X(25).
           02 EMPLOYEE-ADDRESS.
              .
              .
           02 EMPLOYEE-SALARY-DATA.
              03 SALARY-LEVEL                     PICTURE IS 9(02).
              03 SALARY-STEP                      PICTURE IS 9(02).
           02 EMPLOYEE-DEDUCTIONS.
              .
              .
              .
              .
```

Figure 18.15
Table access requiring two subscripts

Print and Read Process

.

.

(reformat data from payroll authorization record
for purposes of printing)

.

.

```
           MOVE SALARY OF SALARY-MATRIX (SALARY-LEVEL,
                                         SALARY-STEP) TO
               SALARY OF OUTPUT-RECORD-DESCRIPTION.
```

.

.

(reformat remainder of data from payroll
authorization record)
Print reformatted data
Read next payroll authorization record

Tables Requiring Three Subscripts

It stands to reason that if COBOL tables requiring the use of two subscripts can be defined, an unlimited number of subscripts may be possible. This is not quite correct. While tables can be defined to use one, two, and now even three subscripts, three is the maximum number of subscripts COBOL table definitions may utilize.

A one-dimensional table is very easy to visualize. It is a table with only one column and at least one row. With only one column, it is unnecessary to specify which column you desire to access. For a column with many rows, a numeric subscript is necessary to indicate which row is needed.

Tables making use of two dimensions can be thought of as matrices. Here the number of both rows and columns is greater than one. Consequently, two subscripts are required when processing a two-dimensional table. The first subscript specifies the row while the second indicates the column within the matrix whose value is being accessed.

Tables with three dimensions are somewhat more difficult to visualize than their one- or two-dimensional counterparts. These tables also require row and column subscripts. In addition, they require a third subscript, which specifies in what plane the desired value can be found.

Assume an organization that utilizes a salary matrix, like the one shown earlier, must produce a summary report that enumerates the numbers of men and women that reside in each cell of the salary table, that is, for instance, how many men are to be found in level one, step three of the organization's payroll structure. A report like this could be produced as a by-product of the report program shown in figure 18.15. Every time a payroll authorization record is read from the input file, a frequency count for the specified salary level and step is incremented by one. However, each payroll authorization record must also specify the gender of the employee, as shown in figure 18.16.

As the processing logic shown in figure 18.16 illustrates, every input record is examined to determine the gender of the employee to whom the record belongs. If the employee is male, the frequency of males (i.e., the third subscript value of one) for that particular combination of salary level and step is incremented by one. Likewise, if the payroll authorization represents a female employee, the second plane (i.e., the third subscript value of two) is incremented by one instead. It also bears noting that the logic shown above only accumulates the frequencies. Logic elsewhere in the program must extract and print these frequencies. A similar program is the object of an exercise at the end of this chapter.

Figure 18.16
Table access requiring two and three subscripts

```
       01  PAYROLL-AUTHORIZATION-RECORD.
           02  EMPLOYEE-NAME                PICTURE IS X(25).
           02  EMPLOYEE-ADDRESS.
               .
               .
           02  EMPLOYEE-GENDER              PICTURE IS X(01).
               88  MALE-EMPLOYEE            VALUE "M".
               88  FEMALE-EMPLOYEE          VALUE "F".
           02  EMPLOYEE-SALARY-DATA.
               03  SALARY-LEVEL             PICTURE IS 9(02).
               03  SALARY-STEP              PICTURE IS 9(02).
           02  EMPLOYEE-DEDUCTIONS.
               .
               .
       01  GENDER-FREQUENCY-COUNT-TABLE.
           02  DATA-PORTION                 PICTURE IS X(336)
                                            VALUE ALL ZERO.
           02  EACH-SALARY-LEVEL            REDEFINES DATA-PORTION
                                            OCCURS 07 TIMES.
               03  EACH-SALARY-STEP         OCCURS 06 TIMES.
                   04  EMPLOYEE-FREQUENCY   OCCURS 02 TIMES
                                            PICTURE IS 9(04).
```

Print and Read Process

.

.

(reformat data from payroll
authorization record
for purposes of printing)

.

.

The frequency table for gender should be examined closely. Actually, each combination of salary level, step, and gender has been allocated a four-digit cell in which frequencies are to be accumulated. How many cells are in this table? The answer is simple. It contains seven rows, six columns, and two planes. Thus it contains ($7 \times 6 \times 2 = 84$) eighty-four cells. How many storage locations can be allocated and initialized in this table? It contains eighty-four cells, each four digits in length, so the total number of storage locations is 336.

Summary

Table definition and subsequent manipulation constitutes a significant portion of what goes on in business data processing. Not only must beginning programmers know how to construct and process relatively simple tables, but they must also be familiar with more complex issues as well.

Managers often insist that slow-executing programs involving significant amounts of table processing be made more efficient. An earlier chapter pointed out that certain improvements can be made by using subscripts. These subscripts require fewer conversion operations each time those subscripts are involved in a program-controlled operation. Furthermore, it was also noted that the SEARCH verb offers greater

```
           MOVE SALARY OF SALARY-MATRIX (SALARY-LEVEL,
                                         SALARY-STEP) TO
                 SALARY OF OUTPUT-RECORD-DESCRIPTION.
           .
           .
```

(reformat remainder of data from payroll authorization record)

.

.

```
           IF MALE-EMPLOYEE
               THEN
                   ADD 1 TO EMPLOYEE-FREQUENCY (SALARY-LEVEL,
                                                SALARY-STEP,
                                                1).
           IF FEMALE-EMPLOYEE
               THEN
                   ADD 1 TO EMPLOYEE-FREQUENCY (SALARY-LEVEL,
                                                SALARY-STEP,
                                                2).
```

.

.

Print reformatted data

Read next payroll authorization record

improvements in program clarity, control, and efficiency than other means of table processing. However, when the tables in a sluggish program already use the SEARCH verb and efficient subscript processing techniques, programmers must use the binary search.

Conventional means of table examination involve a serial process. Where a match between a table entry and the item being compared is guaranteed, then, on the average, half of all rows in the table will be searched each time a SEARCH is undertaken. When a match is not guaranteed, the number of comparison operations may increase significantly. A nonserial processing method makes fewer comparison operations necessary.

The binary search is actually a method whereby the table is successively partitioned into smaller halves. In each instance, the half in which a match is to reside, if one exists, is again split into halves. This partitioning process results in far fewer comparison operations than the more conventional serial search.

The binary search can be used automatically in COBOL programs provided the programmer does two things. First, the OCCURS clause must immediately be followed by the ASCENDING KEY IS clause. Optionally, a DESCENDING KEY IS clause may be used as well. In any event, this clause informs the COBOL compiler which multi-valued variable is to be used for guiding the partitioning process. This requires that the entries in the table actually be arranged either ASCENDINGly or DESCENDINGly.

Second, the SEARCH verb must be modified to contain the word *ALL*. ALL informs the COBOL compiler that the search logic to be generated is the binary search. Also, unlike all earlier uses of the SEARCH verb, the subscript is not initialized prior to the SEARCH with the SET verb. The binary version of the SEARCH provides the subscript with its initial value automatically.

Statement Format

```
12
↓
SEARCH ALL repeateditem
  [AT END statement]
          WHEN  condition.
```

It also happens that some programmers choose to create tables containing homogeneous data rather than combinations of different data. The programmer may find it easier or clearer to manipulate homogeneous tables.

When multiple, homogeneous tables require synchronized access, the SEARCH verb must be instructed to VARY the subscript of the other table involved. Inclusion of the VARY clause in the SEARCH verb directs the computer to insure that different subscripts contain the same row number even though they are to be used in accessing different tables. Thus, it is possible with a single instruction to cause two different subscripts to point to, for instance, row 1 of their respective tables. This particular feature, unfortunately, may not be used with a binary search.

Finally, it is not unusual to have tables requiring the use of more than one subscript. In this instance, the tables in question must be defined with multiple OCCURS clauses. A one-to-one correspondence exists between the number of OCCURS clauses a table possesses and the number of subscripts required to access a variable within that table. This correspondence must be followed. Thus, if a variable is subordinate to three OCCURS clauses, access to a specific value for that variable requires the use of three subscripts.

The order in which subscripts appear in an access statement is always row followed by column followed by plane. Of course, one- and two-dimensional tables do not have planes and do not require the use of a third subscript. Further, one-dimensional tables do not need a column subscript because one-dimensional tables possess only one column.

Notes

1. This assumes an input file whose records feature equal frequencies for each of the fifty states.

Questions for Review

1. If a table contains 100 rows in a processing application where a match is guaranteed for each record to be read, is it much more efficient to use a binary search rather than a serial search? How many comparison operations would be required for a serial search? A binary search?
2. Why must the entries in a table be arranged in strict ascending sequence or descending sequence if a binary search is to be used?
3. Is it possible to improve upon the efficiency of a serial search so that a binary search may not be necessary? Explain.
4. Why is it, when using a binary search, that the SET verb need not be used prior to invoking the SEARCH verb?
5. What are homogeneous tables? What problems arise when a SEARCH verb is used to determine in which row of one table a match has been found, while the corresponding row of another table must then be accessed? How can this problem be overcome? Is this situation a problem in which tables not using the SEARCH verb are used? Explain.
6. What is the maximum number of dimensions a table may possess in COBOL? How many subscripts are required to access a variable subordinate to one OCCURS clause? Two? Three? Explain.
7. How do you determine how many storage units a table is to be allocated and, possibly, initialized?

Programming Assignment

1. Kern Industries markets thermopane window products nationwide. Kern has segmented the continental United States into twelve sales zones. Each zone has its own sales force and manager. In order to spur sales income, the vice-president for marketing at Kern has developed a sales promotion contest in which the zone that produces the largest monthly sales income will receive bonuses to be allocated to the winning zone's sales force in a manner to be determined by the manager for that zone.

   Kern's data processing staff has been directed to construct a computer program to analyze the monthly sales report file for the purpose of producing a report like that shown in figure 18.17. The input to produce this report is as follows:

Record Position	*Data Type*	*Variable*
01–09	N	Salesperson identification
01–02	N	Zone assignment
03–04	N	Division within zone
05–09	N	Sales sequence number
10–11	AN	Department of goods sold
12–14	N	Quantity of goods sold
15–20	N	Unit price (dollars and cents)
21–80	NA	Unused

Figure 18.17
Kern Industries zone sales report

PROGRAMMER ______
PROBLEM ______
DATE ______

REPORT LAYOUT FORM

PAGE ____ OF ____

```
                                                     KERN INDUSTRIES

                                               SALES PROMOTIONAL CAMPAIGN
                                                        ANALYSIS

                                ANALYSIS OF SALES FOR ZONE 01          $ 94,382,101.36

                                     DIVISION 01    $   2,111,848.00          ** WINNER OF ZONE 01 CONTEST **
                                              02          844,210.00
                                              03          557,794.11

                                              09        1,892,411.11

                                ANALYSIS OF SALES FOR ZONE 02          $ 20,644,998.86

                                     DIVISION 01    $     947,123.00
                                              02          886,741.00
                                              03          891,667.66

                                              07        1,003,241.00          ** WINNER OF ZONE 02 CONTEST **

                                ANALYSIS OF SALES FOR ZONE 03          $ 36,441,643.21

                                ANALYSIS OF SALES FOR ZONE 12        $ 96,661,000.88    <--- CONTEST WINNING ZONE

                                     DIVISION 01    $   3,457,633.22

                                              10        4,498,138.11          ** WINNER OF ZONE 12 CONTEST **
```

Modular Construction of COBOL Programs

19

Chapter Outline

Computer software has a life in that it has a beginning, a middle, and an end. When software is initially developed by an organization's team of analysts and programmers, it can be a long and difficult process. The trade literature often has horror stories about software development projects that took longer to complete than originally anticipated, cost far more than had been projected, and failed to perform all of the functions promised at the outset.[1]

Often the persons involved in the initial development of software entrust their products to other individuals who were not a part of the original development team. For those individuals who are engaged in software maintenance, several observations seem commonplace. First, maintaining software is far more difficult than original development. Second, maintenance costs are likely to exceed the cost of original development.

Software development is not an easy task, however. Deadlines must be met and programs must work as expected. Yet a continuity of sorts can be found in initial development. Even if a programmer is working on several programs concurrently, those programs are quite familiar to the programmer involved. Programs under development may be so familiar to their creators that they are difficult to leave mentally or physically at the end of the workday. Programs requiring maintenance are another matter.

It is helpful to understand the environment of program maintenance. First, to require maintenance, a program must have entered regular usage, or production status, as it is termed. A program in production status is transforming inputs into expected outputs for a client. Second, when maintenance is required, it is usually because during its production usage something unanticipated or undesired has occurred. Perhaps the program has ceased functioning, or has continued functioning in a rather strange manner, producing equally strange results. In any event, someone expecting the normal output is not receiving it. Third, for reasons far too involved to explore here, it often seems that programs only require maintenance between midnight and morning on the weekend. It also seems that some unknown, but tremendously important, processing sequence is always disturbed by this software failure. The importance of getting the program running again requires that someone be brought in to correct the problem as soon as possible.

To make matters somewhat worse we must add another law. This law states, with great editorial license, that software is more organized and less confusing to the reader, and least likely to fail the closer it is to its original development. Thus, the patches and fixes added to cure a specific malady contribute to the decay of the program. It appears that the whole software maintenance effort can only lead to the eventual demise of the software being maintained.

What can be gained from the foregoing discussion? First and foremost, software should be designed to minimize the need for subsequent maintenance. Maintenance causes additional maintenance. A second, but somewhat more subtle, observation is that initial development has not been concerned with reducing maintenance all of these years.

Three factors contribute to the attitudes about maintenance during initial development. First, the stress of deadlines and multiple assignments causes programmers to think not of minimizing possible future maintenance costs, but of definite, current-development slippage. Second, programming is still much more of an art than a science. Third, software development organizations have seldom been judged on their ability to produce programs requiring low levels of maintenance.

In this chapter you will be introduced to a special form of modular programming that is possible in COBOL. This version of modular programming requires that programs be broken down into their individual functions and these functions be implemented as separately compilable modules.

As we shall discuss, when programs can be divided into separate isolated modules, the impact of program changes on the entire program can be minimized.

Separately Compilable Modules—What Are They?

In order to comprehend what separately compilable modules are, we must first learn what they are not. Examine the familiar COBOL program in figure 19.1. The 80–80 listing program shown is modular in construction. However, the modules are said to be *internal modules.*

COBOL programs can be made up of two types of modules. The first type, the internal module, is named by a section such as MAIN-CONTROL or PRINT-READ-PROCESS. Sections other than MAIN-CONTROL are invoked by a PERFORM instruction. MAIN-CONTROL is not invoked by a PERFORM because it is the section that assumes control of the computer processor when the program is chosen for execution by the computer system. Another attribute of internal modules is that the program that contains them cannot be successfully compiled if one or more of its internal modules is missing.

Still another attribute of internal modules regards the scope, or impact, of program changes. It is possible to make changes to a section without causing unanticipated errors. It is more likely, however, that changes will cause errors. For instance, what would happen to the program in figure 19.1 if the DISPLAY verb in the second module were changed to a WRITE verb? The result would be catastrophic. Where are the OPEN and CLOSE verbs? What about the required SELECT clauses and the file definitions (FD)?

Internal Module
A COBOL section in the PROCEDURE DIVISION invoked by use of a PERFORM verb. The section contains a function better represented as an isolated entry in the source program.

Page 1 of 2

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    L8080.
    SKIP3
ENVIRONMENT DIVISION.
    SKIP3
DATA DIVISION.
    SKIP1
WORKING-STORAGE SECTION.
01  RECORD-IMAGE               PICTURE IS X(80).
    SKIP3
PROCEDURE DIVISION.
    SKIP2
MAIN-CONTROL SECTION.
    SKIP1
M-C-001.
    ACCEPT RECORD-IMAGE.
    SKIP1
M-C-002.
    PERFORM PRINT-READ-PROCESS
        UNTIL
            RECORD-IMAGE = SPACES.
    SKIP1
M-C-003.
    STOP RUN.
```

Page 2 of 2

```
    SKIP3
PRINT-READ-PROCESS SECTION.
    SKIP1
P-R-P-001.
    DISPLAY RECORD-IMAGE.
    ACCEPT RECORD-IMAGE.
```

Figure 19.1
80-80 listing program

External Module
Once again, the module contains a function better represented in isolated form. Invoked by use of a CALL verb. Contains complete IDENTIFICATION, ENVIRONMENT, DATA, and PROCEDURE DIVISIONS.

The second type of module is said to be an *external module.* Figure 19.2 illustrates the 80–80 listing program restructured to utilize external rather than internal modules.

An examination of the program in figure 19.2 reveals the use of a new instruction—CALL. The CALL verb is much like the PERFORM. Both instructions cause execution of the program to be redirected to the place specified. Control is returned to the statement immediately following either the PERFORM or CALL. The PERFORM invokes internal modules that are a physical part of the program. The CALL, on the other hand, invokes a separately compiled COBOL program module. As figure 19.2 illustrates, three separately compiled COBOL program modules exist. The first is called L8080, the second is READER, and the third is WRITER.

Statement Format

```
12
↓
CALL modulename USING variablelist.
```

Knowing only how the 80–80 program is to function, we can infer that READER, when invoked, returns an eighty-column record image in the variable named RECORD-IMAGE. Thus, READER appears to be a direct replacement for the ACCEPT verb in figure 19.1. We can further infer that WRITER is a direct replacement for the DISPLAY instruction. Every time WRITER is invoked, the record image in RECORD-IMAGE will be displayed upon the printer.

Figure 19.2
80-80 listing program in external module form

Page 1 of 2

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.    L8080.
           SKIP3
       ENVIRONMENT DIVISION.
           SKIP3
       DATA DIVISION.
           SKIP2
       WORKING-STORAGE SECTION.
       01  RECORD-IMAGE                PICTURE IS X(80).
           SKIP3
       PROCEDURE DIVISION.
           SKIP2
       MAIN-CONTROL SECTION.
           SKIP1
       M-C-001.
           CALL "READER" USING RECORD-IMAGE.
           SKIP1
       M-C-002.
           PERFORM PRINT-READ-PROCESS
               UNTIL
                   RECORD-IMAGE = SPACES.
           SKIP1
       M-C-003.
           STOP RUN.
```

Page 2 of 2

```
           SKIP3
       PRINT-READ-PROCESS SECTION.
           SKIP2
       P-R-P-001.
           CALL "WRITER" USING RECORD-IMAGE.
           CALL "READER" USING RECORD-IMAGE.
```

Also obvious by now is that neither the READER nor the WRITER program modules appear in figure 19.2. What do they look like? We know they are both separately compilable COBOL programs. This means that each has an IDENTIFICATION, ENVIRONMENT, DATA, and PROCEDURE DIVISION. What we do not yet know is whether READER accomplishes its reading task by using the ACCEPT verb or the more conventional READ verb. Equally unclear is whether WRITER uses the familiar DISPLAY verb or the more popular WRITE verb instead. The point here is that the program L8080 does not care which of these variations is used. All L8080 requires is that READER reads record images one at a time and that WRITER prints whatever happens to be in RECORD-IMAGE.

The implications of external modules are quite overwhelming. First, changes in any one module are isolated to that module. No other module need be aware of how the other performs its assigned tasks. It makes no difference to L8080 whether READER uses ACCEPT or the READ instruction. Second, each module is separately compiled by the COBOL compiler, so variable, paragraph, and section names need not be unique. The same paragraph name can be used in all three modules, for example. Yet because each module is separately compiled by the COBOL compiler, each program starts with a clean slate when its compilation begins.

Characteristics and Structure of External Modules

Examine the READER module shown in figure 19.3. It appears to be a complete COBOL program featuring all four divisions.

It is important to recognize that the name of this separately compiled and complete COBOL program is READER. This is the same name used in the CALL verb shown in figure 19.2. The names must always be identical. So, even though the WRITER module has not yet been revealed, we know the name in its PROGRAM-ID entry must be WRITER.

Next, separately compiled modules invoked using the CALL verb possess a section within the DATA DIVISION not seen before. The name of the new entry is the LINKAGE SECTION. The LINKAGE SECTION contains variable definitions of the variables lent to the module. The concept of lending is very important to understand.

As figure 19.2 illustrates, RECORD-IMAGE is a variable belonging to module L8080. RECORD-IMAGE is defined in L8080's WORKING-STORAGE SECTION, and is eighty characters in length. When L8080 invokes READER with the CALL instruction, READER is informed to carry out its task USING RECORD-IMAGE. That is, L8080 is lending RECORD-IMAGE to READER to use.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    READER.
    SKIP3
ENVIRONMENT DIVISION.
    SKIP3
DATA DIVISION.
    SKIP1
LINKAGE SECTION.
01  RECORD-IMAGE                 PICTURE IS X(80).
    SKIP3
PROCEDURE DIVISION USING RECORD-IMAGE.
    SKIP1
MAIN-CONTROL SECTION.
    SKIP1
M-C-001.
    ACCEPT RECORD-IMAGE.
    GOBACK.
```

Figure 19.3
One possible implementation of the READER module

In READER, RECORD-IMAGE must be defined in the LINKAGE SECTION to inform the COBOL compiler that RECORD-IMAGE does not belong to READER. Instead, RECORD-IMAGE belongs to the module that invoked READER into action. Items in the LINKAGE SECTION are not allocated space in the area of memory belonging to READER. The eighty positions of primary memory allocated to RECORD-IMAGE can be found in L8080, not in READER. As a result, READER will manipulate a variable (RECORD-IMAGE) that does not belong to READER. The variable being used in READER belongs to L8080.

Statement Format

```
8
↓
LINKAGE SECTION.
```

Also noteworthy is that the PROCEDURE DIVISION statement has a new phrase appended to it. Specifically, the phrase USING RECORD-IMAGE reinforces the fact that RECORD-IMAGE does not belong to READER—it is being lent to READER by another module.

Statement Format

```
8
↓
PROCEDURE DIVISION USING variablelist.
```

Finally, the last instruction in READER terminates READER activity. The GOBACK statement causes control in the program to leave READER and return to the module that caused READER to be invoked. Actually, L8080 caused READER to become active. Consequently, control will be returned to one of the two CALL instructions in L8080 which could have caused READER to be invoked.

Statement Format

```
12
↓
GOBACK.
```

Reduced Scope of Changes to Module Structure

Examine the module named WRITER shown in figure 19.4. You might be surprised to see that instead of the DISPLAY verb, more conventional I/O instructions have been used. Does it matter to L8080 whether DISPLAYs or WRITEs are used? No, in fact, the WRITER module shown in figure 19.5 could replace the module shown in figure 19.4 without any change to the functioning of L8080.

Both WRITER modules, shown in figures 19.4 and 19.5, produce identical outputs. L8080 does not know which version is being used or why. Consequently, if L8080 is using the second alternative and a decision is made to use the first alternative, the change would be carried out with no impact whatsoever upon either L8080 or READER.

It should also be noted that in figure 19.4, WRITER possesses both a WORKING-STORAGE SECTION and a LINKAGE SECTION. WORKING-STORAGE still defines variables belonging to the module. In this case FILE-STATUS does belong to WRITER and appears in the WORKING-STORAGE SECTION. Consequently, the COBOL compiler will allocate six storage locations for FILE-STATUS in the area of memory for WRITER. The LINKAGE SECTION still identifies variables used, but not owned, by WRITER.

Page 1 of 2

Figure 19.4
Alternative one—WRITER module

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  WRITER.
    SKIP3
ENVIRONMENT DIVISION.
    SKIP3
INPUT-OUTPUT SECTION.
    SKIP1
FILE-CONTROL.
    SELECT PRINT-FILE ASSIGN TO UR-1403-S-SYSPRINT.
    SKIP3
DATA DIVISION.
    SKIP1
FILE SECTION.
    SKIP1
FD  PRINT-FILE
    LABEL RECORDS ARE OMITTED.
01  PRINT-RECORD                PICTURE IS X(80).
    SKIP2
WORKING-STORAGE SECTION.
01  FILE-STATUS                 PICTURE IS X(06) VALUE
    "CLOSED".
    88 FILE-NOT-OPENED          VALUE "CLOSED".
    SKIP2
LINKAGE SECTION.
```

Page 2 of 2

```
01  RECORD-IMAGE                PICTURE IS X(80).
    SKIP3
PROCEDURE DIVISION USING RECORD-IMAGE.
    SKIP1
MAIN-CONTROL SECTION.
    SKIP1
M-C-001.
    IF FILE-NOT-OPENED
        THEN
            MOVE "OPENED" TO FILE-STATUS
            OPEN OUTPUT PRINT-FILE.
    SKIP1
M-C-002.
    WRITE PRINT-RECORD
        FROM
            RECORD-IMAGE
        AFTER ADVANCING
            1 LINES.
    GOBACK.
```

Figure 19.5
Alternative two—WRITER module

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  WRITER.
    SKIP3
ENVIRONMENT DIVISION.
    SKIP3
DATA DIVISION.
    SKIP1
LINKAGE SECTION.
01  RECORD-IMAGE               PICTURE IS X(80).
    SKIP3
PROCEDURE DIVISION USING RECORD-IMAGE.
    SKIP1
MAIN-CONTROL SECTION.
    SKIP1
M-C-001.
    DISPLAY RECORD-IMAGE.
    GOBACK.
```

Mechanics of Separate Compilations

So far we have seen three different COBOL program modules. We have seen L8080, READER, and WRITER. How do they fit into the same program deck?

Figure 19.6
Program deck for IBM COBOL compiler

Page 1 of 3

```
CBL        <—— marks the start of the first module
       IDENTIFICATION DIVISION.
       PROGRAM-ID.    L8080.
           SKIP3
       ENVIRONMENT DIVISION.
           SKIP3
       DATA DIVISION.
           SKIP1
       WORKING-STORAGE SECTION.
       01  RECORD-IMAGE                PICTURE IS X(80).
           SKIP3
       PROCEDURE DIVISION.
           SKIP1
       MAIN-CONTROL SECTION.
           SKIP1
       M-C-001.
           CALL "READER" USING RECORD-IMAGE.
           SKIP1
       M-C-002.
           PERFORM PRINT-READ-PROCESS
               UNTIL
                   RECORD-IMAGE = SPACES.
           SKIP1
       M-C-003.
```

Using the IBM COBOL Compiler

Installations featuring the IBM COBOL compiler can direct the compiler to compile the three program modules separately in one program deck.[2] The layout of that deck is shown in figure 19.6.

As the deck in figure 19.6 illustrates, each program module is separated by a record image containing the CBL card. The appearance of the CBL informs the COBOL compiler that a new COBOL program module has just been encountered.

Page 2 of 3

```
           STOP RUN.
           SKIP3
       PRINT-READ-PROCESS SECTION.
           SKIP1
       P-R-P-001.
           CALL "WRITER" USING RECORD-IMAGE.
           CALL "READER" USING RECORD-IMAGE.
CBL            <--- marks the start of the second module
       IDENTIFICATION DIVISION.
       PROGRAM-ID.    READER.
           SKIP3
       ENVIRONMENT DIVISION.
           SKIP3
       DATA DIVISION.
           SKIP1
       LINKAGE SECTION.
       01  RECORD-IMAGE                   PICTURE IS X(80).
           SKIP3
       PROCEDURE DIVISION USING RECORD-IMAGE.
           SKIP1
       MAIN-CONTROL SECTION.
           SKIP1
       M-C-001.
           ACCEPT RECORD-IMAGE.
```

Page 3 of 3

```
           GOBACK.
CBL            <--- marks the start of the third module
       IDENTIFICATION DIVISION.
       PROGRAM-ID.    WRITER.
           SKIP3
       ENVIRONMENT DIVISION.
           SKIP3
       DATA DIVISION.
           SKIP1
       LINKAGE SECTION.
       01  RECORD-IMAGE                   PICTURE IS X(80).
           SKIP3
       PROCEDURE DIVISION USING RECORD-IMAGE.
           SKIP1
       MAIN-CONTROL SECTION.
           SKIP1
       M-C-001.
           DISPLAY RECORD-IMAGE.
           GOBACK.
```

Using the WATBOL Compiler

Note the subtle differences when the same program is submitted to the WATBOL COBOL compiler. As figure 19.7 depicts, the programmer omits the CBL cards shown in figure 19.6. Obviously, the WATBOL compiler determines that a new module begins when an IDENTIFICATION DIVISION is encountered. WATBOL, unlike the IBM COBOL compiler, does not permit the use of a CBL card.

Figure 19.7
Program deck for WATBOL compiler

Page 1 of 3

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.  L8080.
           SKIP3
       ENVIRONMENT DIVISION.
           SKIP3
       DATA DIVISION.
           SKIP1
       WORKING-STORAGE SECTION.
       01  RECORD-IMAGE            PICTURE IS X(80).
           SKIP3
       PROCEDURE DIVISION.
           SKIP1
       MAIN-CONTROL SECTION.
           SKIP1
       M-C-001.
           CALL "READER" USING RECORD-IMAGE.
           SKIP1
       M-C-002.
           PERFORM PRINT-READ-PROCESS
               UNTIL
                   RECORD-IMAGE = SPACES.
           SKIP1
       M-C-003.
           STOP RUN.
```

Common Errors and Misunderstandings

Unique Variable, Paragraph, and Section Names

A frequent point of confusion concerning external, separately compilable modules involves the choice of variable, paragraph, and section names. Within a module, such as L8080, each paragraph and section name must be unique. Indeed, 01-level variables must have unique names.

However, both L8080 and READER can contain a MAIN-CONTROL SECTION. The reason that section names can be duplicated is that these sections appear in different COBOL program modules, each of which is separately compiled. Consequently, duplication of paragraph names and variable names between program modules is possible.

Page 2 of 3

```
       PRINT-READ-PROCESS SECTION.
           SKIP1
       P-R-P-001.
           CALL "WRITER" USING RECORD-IMAGE.
           CALL "READER" USING RECORD-IMAGE.
       IDENTIFICATION DIVISION.
       PROGRAM-ID. READER.
           SKIP3
       ENVIRONMENT DIVISION.
           SKIP3
       DATA DIVISION.
           SKIP1
       LINKAGE SECTION.
       01  RECORD-IMAGE                 PICTURE IS X(80).
           SKIP3
       PROCEDURE DIVISION USING RECORD-IMAGE.
           SKIP1
       MAIN-CONTROL SECTION.
           SKIP1
       M-C-001.
           ACCEPT RECORD-IMAGE.
           GOBACK.
       IDENTIFICATION DIVISION.
       PROGRAM-ID. WRITER.
```

Page 3 of 3

```
           SKIP3
       ENVIRONMENT DIVISION.
           SKIP3
       DATA DIVISION.
           SKIP1
       LINKAGE SECTION.
       01  RECORD-IMAGE                 PICTURE IS X(80).
           SKIP3
       PROCEDURE DIVISION USING RECORD-IMAGE.
           SKIP1
       MAIN-CONTROL SECTION.
           SKIP1
       M-C-001.
           DISPLAY RECORD-IMAGE.
           GOBACK.
```

Figure 19.8
Lack of similarity named variables in READER (L8080)

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.  READER.
           SKIP3
       ENVIRONMENT DIVISION.
           SKIP3
       DATA DIVISION.
           SKIP1
       LINKAGE SECTION.
       01  ITEM-TO-PRINT           PICTURE IS X(80).
           SKIP3
       PROCEDURE DIVISION USING ITEM-TO-PRINT.
           SKIP1
       MAIN-CONTROL SECTION.
           SKIP1
       M-C-001.
           DISPLAY ITEM-TO-PRINT.
           GOBACK.
```

Comments Concerning Like Variable Names

As was just noted, L8080 and READER can both use the variable name RECORD-IMAGE; however, they do not have to use the same variable name. Examine the newest version of READER shown in figure 19.8. The variable in the READER LINKAGE SECTION has been given a name other than RECORD-IMAGE.

As this most recent version of READER shows, the area in L8080 that contains a record image is named ITEM-TO-PRINT. A module being lent a variable has the option of naming that variable whatever it chooses. Names are unimportant to proper execution; however, the choice of different variable names may make it more difficult for a future reader of the entire program to understand the workings of the program and its modules. For this reason alone, programmers constructing program modules manipulating lent variables are strongly encouraged to choose the same names as used in the main routine (L8080).

It should be recognized that RECORD-IMAGE and ITEM-TO-PRINT have identical attributes. That is, each variable definition describes an area in primary storage eighty alphanumeric positions in length.

Omitting the USING Clauses

A frequent error made by those unfamiliar with external, separately compilable modules is to omit the USING clauses from both the CALL verb and the PROCEDURE DIVISION. In L8080, if the invocation of READER has omitted the USING clause, L8080 might appear as shown in figure 19.9.

A careful examination of L8080 in figure 19.9 reveals that both READER and WRITER are invoked in the usual manner, with one exception. No CALL verb contains a USING clause. This is legal but it doesn't make any sense. When no USING clause appears in the CALL and/or PROCEDURE DIVISION statement of an invoked module, no variable is available to be shared between, in this instance, L8080 and READER. The shared variable makes it possible for READER to fill that variable with a record image and L8080 to use that record image as it chooses.

In a very few circumstances, it makes sense to invoke a module without also providing that module access to one or more variables owned by the invoking module. The 80–80 listing program does not belong to that set of circumstances. If READER cannot have access to RECORD-IMAGE in L8080, where will the record image read by READER be placed? Without the USING clause in the CALL instruction, READER cannot return a record image to L8080. If L8080 has no record image to show for having invoked READER, why should it invoke WRITER? After all, if READER has no record image, there is no record image to be printed. Consequently, the CALL verbs must contain USING clauses if communication between L8080 and READER, and L8080 and WRITER is to occur.

Page 1 of 2

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.  L8080.
           SKIP3
       ENVIRONMENT DIVISION.
           SKIP3
       DATA DIVISION.
           SKIP1
       WORKING-STORAGE SECTION.
       01  RECORD-IMAGE            PICTURE IS X(80).
           SKIP3
       PROCEDURE DIVISION.
           SKIP1
       MAIN-CONTROL SECTION.
           SKIP1
       M-C-001.
           CALL "READER".
           SKIP1
       M-C-002.
           PERFORM PRINT-READ-PROCESS
               UNTIL
                   RECORD-IMAGE = SPACES.
           SKIP1
       M-C-003.
           STOP RUN.
```

Page 2 of 2

```
           SKIP3
       PRINT-READ-PROCESS SECTION.
           SKIP1
       P-R-P-001.
           CALL "WRITER".
           CALL "READER".
```

Figure 19.9
L8080 in error—missing USING clauses

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.  READER.
           SKIP3
       ENVIRONMENT DIVISION.
           SKIP3
       DATA DIVISION.
           SKIP1
       LINKAGE SECTION.
       01  RECORD-IMAGE            PICTURE IS X(80).
           SKIP3
       PROCEDURE DIVISION.
           SKIP1
       MAIN-CONTROL SECTION.
           SKIP1
       M-C-001.
           DISPLAY RECORD-IMAGE.
           GOBACK.
```

Figure 19.10
READER in error—missing USING clause

USING clauses are also required in the PROCEDURE DIVISION statement for modules invoked by others. Thus the READER PROCEDURE DIVISION statement must contain a USING clause. If not, even though L8080 made RECORD-IMAGE available for READER to use, READER has elected not to use it, as shown in figure 19.10.

READER is now in error. Even though READER contains a LINKAGE SECTION, READER cannot have access to RECORD-IMAGE without the USING clause in the PROCEDURE DIVISION statement. The USING clause must be supplied.

Figure 19.11
READER permitting blank card input and ACCEPT

Page 1 of 2

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.   READER.
           SKIP3
       ENVIRONMENT DIVISION.
           SKIP3
       DATA DIVISION.
           SKIP1
       LINKAGE SECTION.
       01  RECORD-IMAGE.
           02 FIRST-FOUR               PICTURE IS X(04).
           02 FILLER                   PICTURE IS X(72).
           02 LAST-FOUR                PICTURE IS X(04).
           SKIP1
       01  EOF-INDICATOR               PICTURE IS 9(01).
           SKIP3
       PROCEDURE DIVISION USING RECORD-IMAGE,
                                EOF-INDICATOR.
           SKIP1
       MAIN-CONTROL SECTION.
           SKIP1
       M-C-001.
           ACCEPT RECORD-IMAGE.
           IF FIRST-FOUR = ALL "$" AND
               LAST-FOUR = ALL "$"
```

Page 2 of 2

```
               THEN
                   MOVE 1 TO EOF-INDICATOR
               ELSE
                   MOVE 0 TO EOF-INDICATOR.
           GOBACK.
```

Permitted Variations in Defining Lent Variables

Assume for the moment that READER was not permitted to interpret the first blank record encountered as the end of the input file. Indeed, the input file may contain many blank records. If READER is still required to use the ACCEPT verb, a problem arises. How does READER know when it has encountered the last record in the input file? If READER cannot determine which record is the last record, READER risks attempting to read beyond the end of the input file and abnormally terminating.

To remedy this problem, assume each input file to be processed by READER contains dollar signs in both the first and last four positions of the very last record in the file. This requires READER to be modified as shown in figure 19.11.

In this most recent revision to READER, RECORD-IMAGE has been defined as a multi-field variable rather than a single-field variable. This is more than acceptable. What is important is that the overall dimensions of RECORD-IMAGE be consistently defined between L8080 and READER. Both definitions state that RECORD-IMAGE is eighty characters in length.

Lending More Than One Variable

As figure 19.11 reveals, a program module can borrow more than one variable from the module that causes its invocation. When this occurs, each variable must be defined in the borrower's LINKAGE SECTION, and both variables must appear in the USING clauses described earlier. In fact, the variables must appear in the USING clauses in the same order, as shown in figure 19.12.

If the ordering of the variables in the two USING clauses does not agree, unpredictable results will occur. In figure 19.12, READER expects the first variable lent by L8080 to be eighty characters long and alphanumeric. READER expects the second to be one character in length and numeric.

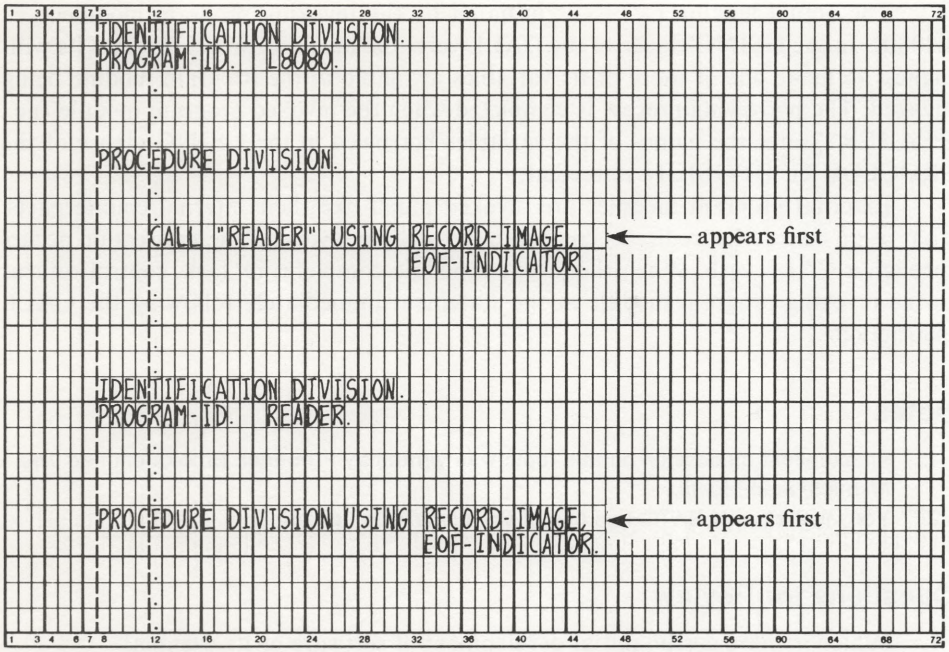


Figure 19.12 Order of multiple lent/borrowed variables in USING clause

Summary

The largest component cost of software is attributable to maintenance. Thus, it would seem that software should be initially designed and constructed to minimize, if not eliminate, maintenance requirements. Unfortunately, the initial development of software is a painful process reflecting the pressures of deadlines, cost overruns, and the like. Consequently, software is initially designed to reflect not the maintenance environment, but the development environment instead.

One way to minimize the future maintenance needs of software is to design more reliable and easier to comprehend software systems. Programs should be broken down into the necessary and obvious functions required. Each function must be easily identified and feature as few linkages with the other functions as possible. Greater separation and fewer linkages provide for less complex program structures.

A method to separate program functions is to represent those functions as external, separately compilable modules. An additional benefit of external modules is that they can be altered without affecting the remainder of the program structure. This is not the case when internal modules are used.

External modules are complete COBOL programs. Each possesses an IDENTIFICATION, ENVIRONMENT, DATA, and PROCEDURE DIVISION. These modules communicate with one another by lending or borrowing the variables required to complete their assigned tasks. For instance, L8080 lent READER the variable RECORD-IMAGE so READER could place a record image into RECORD-IMAGE on the behalf of L8080.

The borrowing modules define the variables they have been lent in a new section known as the LINKAGE SECTION. Variables can appear in any order within the LINKAGE SECTION. Furthermore, variables defined in this section cannot possess the VALUE clause. The reason is that variables may be assigned initial values only by the modules that own them—not by modules that borrow them.

The actual lending process is initiated by the invoking program module (i.e., L8080). In the USING clause of the CALL verb, the invoking program states its willingness to make one or more of its variables available to the program module being invoked. The borrowing process is completed when the invoked program module (i.e., READER) states the names of the variables borrowed in the USING clause of the PROCEDURE DIVISION statement.

External program modules are invoked much like their internal counterparts. That is, an instruction redirects control to the module invoked. The redirection occurs through a CALL verb. When the invoked module completes its task, control returns to the invoking program module at the statement immediately following the CALL instruction.

Notes

1. An excellent discussion on this topic was written by Frederick Brooks entitled *The Mythical Man-Month.*
2. You must specify BATCH as one of the compiler's options.

Questions for Review

1. Which is more costly, initial development of software or its eventual maintenance? Why?
2. Why would breaking down a program's functions into separately identifiable modules improve the reader's ability to understand the program's task and method of operation?
3. Which type of modules, internal or external, provide the greatest degree of module separation? What is meant by separation?
4. Describe both the parallels and the differences between the PERFORM and CALL instructions.
5. Can program modules be lent more than one variable? In cases where more than one variable is lent, must the variables be lent in the same order in which they were borrowed? Must they appear in the same order in the borrower's LINKAGE SECTION?

Problems

1. Examine the following program modules very carefully. What, if anything, is wrong? Assume the entire program is to be compiled by the WATBOL COBOL compiler.

```
CBL
       IDENTIFICATION DIVISION.
       PROGRAM-ID. L8080.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  EOF-INDICATOR              PICTURE IS 9(01).
           88 END-OF-FILE             VALUE 1.
       01  RECORD-IMAGE               PICTURE IS X(80).
       PROCEDURE DIVISION.
       MAIN-CONTROL SECTION.
       M-C-001.
```

```
           CALL "READER" USING RECORD-IMAGE,
                               EOF-INDICATOR.
           PERFORM PRINT-READ-PROCESS
               UNTIL
                   END-OF-FILE.
           STOP RUN.
       PRINT-READ-PROCESS SECTION.
       P-R-P-001.
           CALL "WRITER" USING RECORD-IMAGE.
           CALL "READER" USING RECORD-IMAGE.
                               EOF-INDICATOR.
 CBL
       IDENTIFICATION DIVISION.
       PROGRAM-ID.  READER.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       LINKAGE SECTION.
       01  RECORD-IMAGE.
           02 FIRST-FOUR       PICTURE IS X(04).
           02 FILLER           PICTURE IS X(72).
           02 LAST-FOUR        PICTURE IS X(04).
       01  EOF-INDICATOR       PICTURE IS 9(01).
       PROCEDURE DIVISION USING RECORD-IMAGE,
                                EOF-INDICATOR.
       MAIN-CONTROL SECTION.
       M-C-001.
           ACCEPT RECORD-IMAGE.
           IF FIRST-FOUR = ALL "$" AND
              LAST-FOUR  = All "$"
             THEN
                 MOVE 1 TO EOF-INDICATOR
             ELSE
                 MOVE 0 TO EOF-INDICATOR.
           GOBACK.
 CBL
       IDENTIFICATION DIVISION.
       PROGRAM-ID.  WRITER.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       LINKAGE SECTION.
       01  RECORD-IMAGE        PICTURE IS X(80).
       PROCEDURE DIVISION USING RECORD-IMAGE.
       MAIN-CONTROL SECTION.
       M-C-001.
           DISPLAY RECORD-IMAGE.
           GOBACK.
```

Programming Assignments

1. Revise the Gordon Information Services assignment from chapter 17 so that the program calls upon a routine named READER to access records from the input file. The READER module is to be passed only one parameter. That parameter is the record description that is to hold the record read from the input file. If, upon return to the MAIN routine, the parameter contains all blanks, MAIN will assume an end of file has been reached and no more input records remain to be read. On the other hand, a nonblank record indicates the invocation of READER has brought forth yet another input record to be processed.

2. Revise the Johnston Mailing Services assignment from chapter 17 so that it invokes a module named WRITER. WRITER prints each of the detail lines that appear on the program's only printed report. In fact, the main program must not contain any SELECT, FD, OPEN, CLOSE, or WRITE statements to produce the report. All such statements are to be located within WRITER. WRITER is to be passed two parameters when invoked. The first is the line to be printed. The second is a one-character variable (PICTURE IS X.) containing a blank if a detail line is to be printed, or a nonblank if a detail line is not to be printed. In the case of the former, additional printing requests will be made of WRITER by MAIN. The latter, however, indicates WRITER is to print its report trailers and close the print file before returning control to MAIN. No further printing requests of WRITER will be made by MAIN.

Appendixes

Appendix A
Constructing COBOL Programs: Antibugging and Debugging

There are eight steps in constructing working programs using the COBOL language. They are logical assessment, report declarations, primary record declarations, translation of the pseudo-code, review of the source program, data entry/program preparation, correction of compile-time errors, and correction of execution-time errors.

Logical Assessment

An old adage states "an ounce of prevention is worth a pound of cure." A beautifully formatted COBOL program, no matter how pleasing to the reader, is worth little if its underlying logical structure is faulty. Therefore, the actual beginning of the programming process starts with devising a logically correct solution to the programming problem. Too often beginning programming students rush to the keypunches or interactive terminals so that they may begin working on their COBOL programs. However, the best beginning for a program is to identify whether the program is an edit, report, or file maintenance task.

We know that programs can be classified according to the major task that each program is to accomplish. We also know that each major program type shares a common logical structure. This structure dictates the order in which the program is to process its data to produce the desired results. Examine the description of edit programs shown in figure A.1.

Figure A.1
Basic edit model

```
Statement              Pseudo-coded statement
Number
  1                    Read the very first record to be
                           checked field by field
  2                    Perform the field-by-field edit
  2a                       until
  2b                           the file is empty
  3                    Stop the program's execution

  4          Field-by-Field Edit
  5                    Print the record just read
  6                    If field 1 is invalid
  6a                       then
  6b                           Print appropriate error message
  7                    If field 2 is invalid
  7a                       then
  7b                           Print appropriate error message
  8                    Read next record to be checked
```

The logic shown in figure A.1 is a very simple edit in which only two fields need to be examined. Nonetheless, it illustrates that all edit programs, irrespective of the data processing application involved, adhere to this basic model.

So you, as a programming student, should first find the model that describes the data processing application for which you are to construct a COBOL program. Then, using that model as the basis of your program logic, add, alter, and/or delete those operations required for your assignment. You do not need to start from scratch. A model exists for the program you wish to construct. Study that model and learn how it functions. You cannot expect a COBOL program to work properly if you do not understand its underlying structure.

Report Declarations

Every computer program produces some form of printed output. These outputs, or reports as they are more commonly known, feature title lines that will appear at the top of each page, detail material that will appear in the middle of the page, and, possibly, trailer lines that will appear at the bottom of each page. These title, detail, and trailer entries require their own 01-level record descriptions.

Converting any line entry in a report to a COBOL declaration for the WORKING-STORAGE SECTION is not difficult if it is done carefully. Some entries can be defined as a single, nonnumeric literal without the use of subordinate fields (i.e., 02, 03, . . . , levels). Still other line entries in the report must be defined as record structures that will permit access to fields within the record. Whatever format is required, these line entries must be defined and checked carefully.

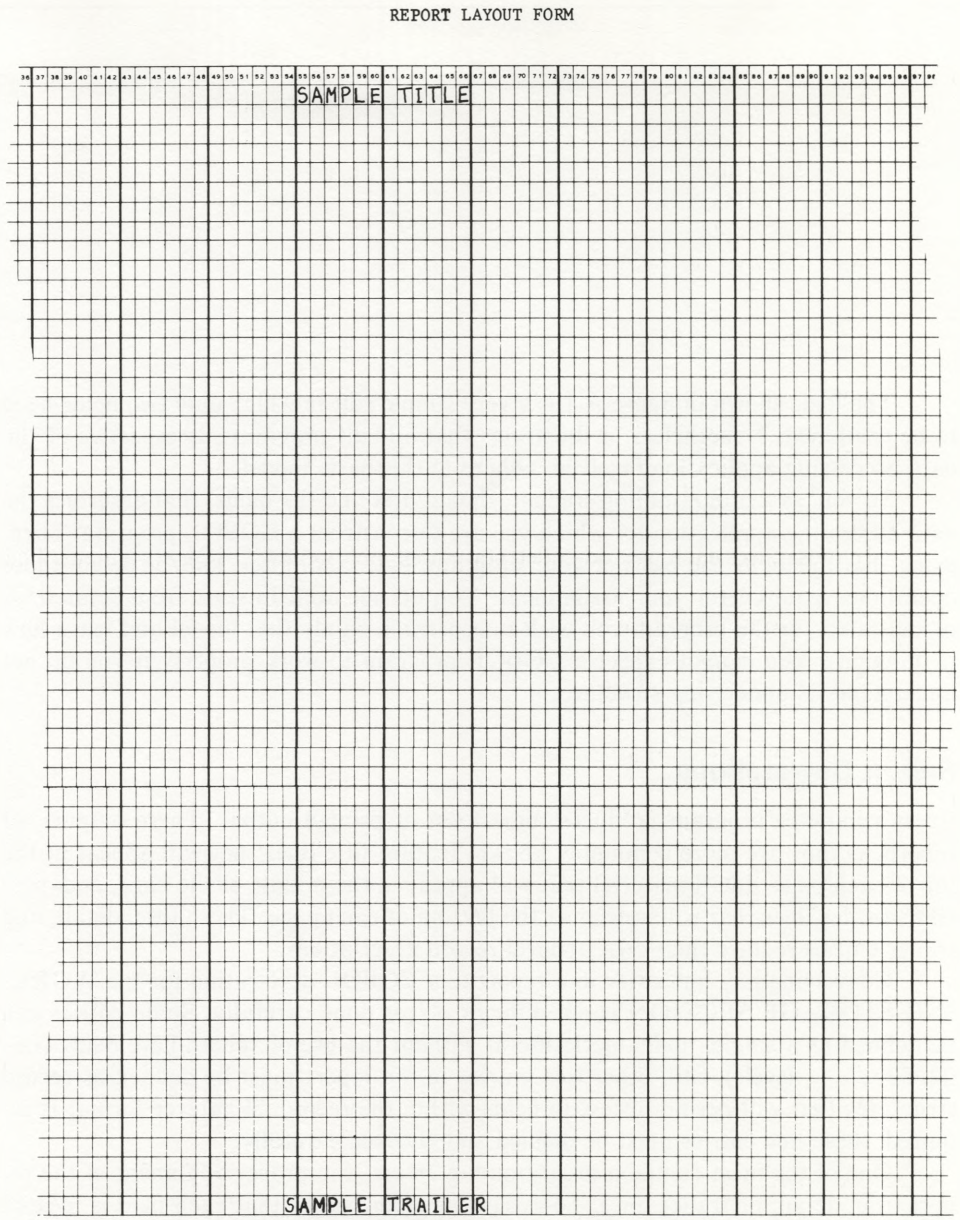
One of the most common programming errors is improper alignment of the report title, detail, or trailer lines. To avoid this error, verify that each field is located precisely where it should be, using either simple graph paper or a COBOL coding form. Two sheets of graph paper can be taped together so that each line and column of a complete report page is represented.

Figure A.2 is a very simple report with two basic entry types. Knowing on what lines and in which columns printed materials are to appear will greatly ease the construction of the 01-level record structures.

After the programmer knows where and how each title, detail, and trailer entry is to appear, it is necessary to translate those entries into COBOL record structures. COBOL coding forms that clearly show the locations of the A- and B-Margins are of significant assistance in properly constructing these items. (See figure A.3.)

Constructing record definitions at a data entry device without prior preparation is unsatisfactory and guaranteed to produce record descriptions that will need to be corrected.

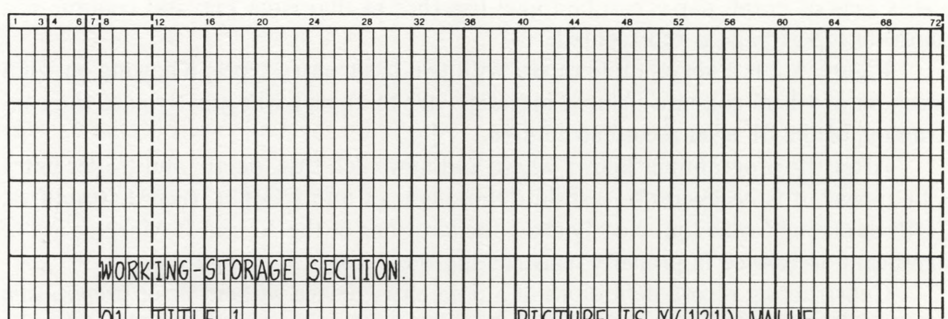
Figure A.2
Report layout form


REPORT LAYOUT FORM

SAMPLE TITLE

SAMPLE TRAILER

Figure A.3
COBOL coding form


```
       WORKING-STORAGE SECTION.
       01  TITLE-1                            PICTURE IS X(121) VALUE
           "                                                   SAMPLE T
      -    "ITLE".
       01  TRAILER-1                          PICTURE IS X(121) VALUE
           "                                                  SAMPLE TR
      -    "AILER".
```

Primary Record Declarations

All computer programs operate on input. What does that input look like? How does the COBOL program need to access the data within each input record?

The logic model that guides the development of the COBOL program answers these questions. Once you know how the input data are to be used, it is possible to use a coding form to construct an appropriate input record description. Remember, the coding form will help you insure that the A- and B-Margin entries are properly made and that the appearance of the record structure is uniform.

Translation of the Pseudo-Code

At this point in the process, you have identified all record and variable declarations and have recorded them on a COBOL coding form. The WORKING-STORAGE SECTION is complete. The program model is clearly understood by you. Now you must translate each of the pseudo-coded instructions that appear in the model into COBOL.

In most cases each pseudo-coded statement yields one COBOL statement. There are times, however, when there is not a one-to-one correlation. In any event, each statement should be recorded on the COBOL coding form, again to insure that the appearance of the program remains consistent.

Remember that the PROCEDURE DIVISION includes program comments as well as section names, paragraph names, and COBOL statements. These comments explain the functions that are taking place to a reader. Unfortunately, too many student programmers do not include comments during the initial construction of the program. Students fail to recognize that they too are readers of the program. Comments are required if the desk checking is to yield the desired results. Comments are *not* frills to be added to a program after it works properly. They are an integral part of the program that must be present from the outset.

Appendix C contains a summary of the syntax of the major COBOL statements discussed in this text. Before using any COBOL statement, consult this summary to insure that you properly use elements of the COBOL language. Obviously, you will use some statements so often that you will not need to check them. However, continue to check those about which you have any doubt. If your questions about a COBOL statement go beyond issues of syntax, consult the index.

Review of the Source Program

Too many students run for the data entry devices when they finish writing their source programs. It is far better to carefully verify that the written version of your COBOL program will (1) be translated by the COBOL compiler without difficulty, and (2) transform all input data into the precise outputs desired. This last stage of verification prior to data entry is known as desk checking.

As the name implies, desk checking a program means examining the program, keeping in mind the operation of the COBOL compiler and the computer. This is the final check of a program that should have been examined carefully throughout its construction. The examination of each step and the desk checking will lessen considerably the likelihood of serious error.

One final point. Many student programmers are eternal optimists who hope that thorough desk checking of their programs is not necessary and that their programs will function perfectly (or at least nearly so) the first time they are submitted for processing. These programs don't, however. Consequently, students end up examining their programs again and again to correct each of the many errors. So, if you end up desk checking anyway, why not make a mad dash for the data entry devices? The answer is simple. Programmers who thoroughly verify the correctness of their programs before

data entry have fewer data entry activities to perform. Their programs contain fewer errors and, thus, require far fewer corrections. They complete their programs sooner. They are less likely to lose their sanity and common sense than individuals who forsake verification so that they may be among the first to claim a data entry device. These individuals require more time to get their programs operational, if it is even possible to do so. The moral: the time spent verifying program correctness results in an operational program in the least amount of time. An ounce of prevention *is* worth a pound of cure.

Data Entry/Program Preparation

If all goes well after you have verified the correctness of your COBOL program, you will want to transfer your COBOL program to a machine-readable medium. Some computer systems still feature punched-card input in which programs must be keypunched and read by a card reader. Other systems offer video display terminals in which programs can be entered into a disk file that will belong to the programmer.

Your instructor will tell you which medium you will use and, possibly, how you are to use it. These instructions will also identify which control statements you must supply to the computer system so that the COBOL compiler will translate your program.

Correction of Compile-Time Errors

Some errors, such as errors in syntax, cannot be avoided no matter how hard you try. Even if you write the COBOL record descriptions in the proper margins on the coding form, it is possible to make a typing error during data entry. When this happens, the COBOL program will produce warning/error messages that note errors in syntax.

Warning and error messages produced by COBOL compilers have earned a reputation for being difficult to understand. In fact, in some circumstances, it may seem that the designers of the COBOL compiler deliberately chose wordings for error messages that obscure the nature of the error. However, in an overwhelming number of instances, the messages produced by the COBOL compiler do contain ample informatory material. Note the error messages in figure A.4.

Figure A.4 illustrates two warning messages ("-W") that are quite common in COBOL programming environments. This program contains two statements that were started in the wrong margin. Statement 6, although not shown, is the DATA DIVISION statement required by every COBOL program. Inadvertently, the word *DATA* was started in the B- rather than the A-Margin. Statement 193 represents a MOVE instruction that began in the A-Margin rather than in the B-Margin.

Another example of program diagnostics is shown in figure A.5. These also represent common errors. In statement 22 of a COBOL program a numeric variable (PAGE-NUMBER) was mistakenly given an alphanumeric PICTURE clause. As a result, the literal in that definition is of a numeric nature (no delimiting quotes) rather than nonnumeric (with delimiting quotes). Unfortunately, this very same variable is used in an ADD statement in card 179 of the program. Since arithmetic operations require numeric variables, the ADD cannot possibly accept a nonnumeric variable like PAGE-NUMBER. Therefore, the COBOL compiler replaces PAGE-NUMBER in statement 179 with the word TALLY, which is a predefined variable that is a part of every COBOL program. The programmer obviously wants to increment PAGE-NUMBER by one rather than TALLY. Consequently, the definition for PAGE-NUMBER will have to be corrected so that it is a numeric variable, and that will solve the problem in statement 179.

This last example illustrates a subtle point with COBOL. There are instances where a single error can cause warning/error messages in more than one statement of a program. When that one problem is corrected, many warning/error messages may disappear.

```
Card   Error message
6      IKF1086I-W   DATA SHOULD BEGIN A-MARGIN.
193    IKF1087I-W   ' MOVE ' SHOULD NOT BEGIN A-MARGIN.
```

Figure A.4
Sample error messages from an IBM COBOL compiler

```
Number   Statement  ← from the COBOL program
00022    02 PAGE-NUMBER            PICTURE IS 9(03) VALUE 1.
00179    ADD 1 TO PAGE-NUMBER.

Card   Error message      ← produced by the COBOL compiler
22     IKF2128I-C   VALUE CLAUSE LITERAL DOES NOT CONFORM
                    TO PICTURE. CHANGED TO BLANKS.
179    IKF4019I-E   DNM=1-251 (AN) MAY NOT BE USED AS
                    ARITHMETIC OPERAND IN ADD STATEMENT.
                    ARBITRARILY SUBSTITUTING TALLY.
```

Figure A.5
Additional COBOL diagnostics

```
Number   Statement  ← from the COBOL program
00022    02 PAGE-NUMBER            PICTURE IS 9(03).
00179    ADD 1 TO PAGE-NUMBER

Diagnostic output ←—— produced by the COBOL compiler
PROGRAM                 RPT0701
LAST PSW BEFORE ABEND = FF750000302B0C84        SYSTEM COMPLETION CODE = 0C7
LAST CARD NUMBER/VERB NUMBER EXECUTED -- CARD NUMBER 000179/VERB NUMBER 01
                                                          FLOW TRACE
RPT0701   000166 000167 000195 000166 000167 000195 000166 000167 000195
          000166 000167 000195 000166 000167 000195
RPT0701   000166 000167 000195 000166 000167
                                                END OF COBOL DIAGNOSTIC AIDS
```

Figure A.6
Abnormal program termination—data exception

Correction of Execution-Time Errors

Programs that are successfully compiled by the COBOL compiler may not necessarily function without error. A clean compile (i.e., no warning/error messages) indicates that the compiler could not find any errors in syntax or usage. There may be other problems in your program that the compiler is not capable of finding.

The COBOL compiler does not check a program's logic—nor can it. For that reason, a program whose logic is faulty will produce results other than those desired. How do you know if this has happened to you?

Many student programmers sigh with great relief when their programs are compiled without warning or error messages. They seldom examine the output produced to determine whether it is correct. After all, if they look closely, they may discover a problem with the proper functioning of the program. However, program outputs must be checked *line by line* to insure that the program works as intended. If it doesn't, the programmer must determine why.

When reviewing program logic, begin with the pseudo-code. Was it correctly structured? If so, was the COBOL a faithful translation of the pseudo-code? If not, why wasn't this problem detected during desk checking?

There are other types of errors, however, that do not involve logic. Examine the COBOL program segment and resulting diagnostic outputs shown in figure A.6.

This problem is similar to the one shown in figure A.5. Note that a numeric variable has not been assigned an initial value. Yet, according to statement 179, the programmer wants to increment whatever value that variable contains by one. What value does PAGE-NUMBER start with? Is it numeric? The answer is that in the absence of a VALUE clause, we don't know whether or not PAGE-NUMBER contains a numeric value. Chances are excellent, however, that PAGE-NUMBER contains a nonnumeric value. Consequently, the ADD instruction will fail, as will the execution of the program.

The example shown in figure A.6 was produced using the IBM OS/VS COBOL compiler. This particular compiler will, upon request, produce a program that displays the sequence of sections/paragraphs invoked during program execution in the event of an abnormal termination. In this instance, we can trace program execution from statements 166 to 167 to 195 again and again. It appears that the last section/paragraph entered before abnormal termination was statement 00167. In fact, it is statement 00179 that has caused the problem. As the diagnostic information in figure A.6 reveals, the ADD verb in statement 00179 has caused a data exception; that is, it attempted to involve a numeric field with nonnumeric contents in an arithmetic operation.

Not all COBOL compilers will produce programs that provide this much information in the event of an abnormal termination. However, COBOL features an instruction that will provide almost the same information. That instruction, READY TRACE, will produce a listing of every section and paragraph name executed in the order they were invoked. The last section or paragraph named is where the abnormal termination occurred. Of course, you still have to determine which executable statement within that section or paragraph caused the abnormal termination.

READY TRACE can produce volumes of printed output. To help minimize the printed output, there is the additional instruction, RESET TRACE. READY TRACE starts the printing of section and paragraph names within the PROCEDURE DIVISION while RESET TRACE stops that process. The programmer can use these two instructions to cause traces of suspect areas in the PROCEDURE DIVISION rather than examining the entire program.

Both READY and RESET TRACE are instructions that must appear only within the B-Margin of a COBOL program. You can use either as often as necessary.

Appendix B
COBOL Reserved Words

This is the complete list of reserved words for COBOL. Some are not used in this text, which does not discuss *all* of COBOL's features. They are included, nevertheless, for reference. The reserved words that are discussed in this text are listed in the index.

ACCEPT
ACCESS
ACTUAL
ADD
ADDRESS
ADVANCING
AFTER
ALL
ALPHABETIC
ALPHANUMERIC
ALPHANUMERIC-
EDITED
ALTER
ALTERNATE
AND
APPLY
ARE
AREA
AREAS
ASCENDING
ASSIGN
AT
AUTHOR

BASIS
BEFORE
BEGINNING
BLANK
BLOCK
BOTTOM
BY

CALL
CANCEL
CBL
CD
CF
CH
CHANGED
CHARACTER
CHARACTERS
CLOCK-UNITS
CLOSE
COBOL
CODE
COLUMN
COM-REG
COMMA
COMMUNICATION
COMP
COMP-1
COMP-2
COMP-3
COMP-4
COMPUTATIONAL
COMPUTATIONAL-1
COMPUTATIONAL-2
COMPUTATIONAL-3
COMPUTATIONAL-4
COMPUTE
CONFIGURATION
CONSOLE
CONSTANT
CONTAINS
CONTROL
CONTROLS

COPY
CORE-INDEX
CORR
CORRESPONDING
COUNT
CSP
CURRENCY
CURRENT-DATE
CYL-INDEX
CYL-OVERFLOW
CO1
CO2
CO3
CO4
CO5
CO6
CO7
CO8
CO9
C10
C11
C12

DATA
DATE
DATE-COMPILED
DATE-WRITTEN
DAY
DAY-OF-WEEK

DE
DEBUG
DEBUG-CONTENTS
DEBUG-ITEM
DEBUG-LINE
DEBUG-NAME
DEBUG-SUB-1
DEBUG-SUB-2
DEBUG-SUB-3
DEBUGGING
DECIMAL-POINT
DECLARATIVES
DELETE
DELIMITED
DELIMITER
DEPENDING
DEPTH
DESCENDING
DESTINATION
DETAIL
DISABLE
DISP
DISPLAY
DISPLAY-ST
DIVIDE
DIVISION
DOWN
DUPLICATES
DYNAMIC

EGI
EJECT
ELSE
EMI
ENABLE
END
END-OF-PAGE
ENDING
ENTER
ENTRY
ENVIRONMENT
EOP
EQUAL
EQUALS
ERROR
ESI
ETI
EVERY
EXAMINE
EXCEEDS
EXCEPTION
EXHIBIT
EXIT
EXTEND
EXTENDED-SEARCH

FD
FILE
FILE-CONTROL
FILE-LIMIT
FILE-LIMITS
FILLER
FINAL
FIRST
FOOTING
FOR
FROM

GENERATE
GIVING
GO
GOBACK
GREATER
GROUP

HEADING
HIGH-VALUE
HIGH-VALUES
HOLD

I-O
I-O-CONTROL
ID
IDENTIFICATION
IF
IN
INDEX
INDEXED
INDICATE
INITIAL
INITIALIZE
INITIATE
INPUT
INPUT-OUTPUT
INSERT
INSPECT
INSTALLATION
INTO
INVALID
IS

JUST
JUSTIFIED

KEY
KEYS

LABEL
LABEL-RETURN
LAST
LEADING
LEAVE
LEFT
LENGTH
LESS
LIBRARY
LIMIT
LIMITS
LINAGE
LINAGE-COUNTER
LINE
LINE-COUNTER
LINES
LINKAGE
LOCK
LOW-VALUE
LOW-VALUES
LOWER-BOUND
LOWER-BOUNDS

MASTER-INDEX
MEMORY
MERGE
MESSAGE
MODE
MODULES
MORE-LABELS
MOVE
MULTIPLE
MULTIPLY

NAMED
NEGATIVE
NEXT
NO
NOMINAL
NOT
NOTE
NSTD-REELS
NUMBER
NUMERIC
NUMERIC-EDITED

OBJECT-COMPUTER
OBJECT-PROGRAM
OCCURS
OF
OFF
OH
OMITTED
ON
OPEN
OPTIONAL
OR
ORGANIZATION
OTHERWISE
OUTPUT
OV
OVERFLOW

PAGE
PAGE-COUNTER
PERFORM
PF
PH
PIC
PICTURE
PLUS
POINTER
POSITION
POSITIONING
POSITIVE
PREPARED
PRINT-SWITCH
PRINTING
PRIORITY
PROCEDURE
PROCEDURES
PROCEED
PROCESS
PROCESSING
PROGRAM
PROGRAM-ID

QUEUE
QUOTE
QUOTES

RANDOM
RANGE
RD
READ
READY
RECEIVE
RECORD
RECORD-OVERFLOW
RECORDING
RECORDS
REDEFINES
REEL
REFERENCES
RELATIVE
RELEASE
RELOAD
REMAINDER
REMARKS
REMOVAL
RENAMES
REORG-CRITERIA

REPLACING
REPORT
REPORTING
REPORTS
REREAD
RERUN
RESERVE
RESET
RETURN
RETURN-CODE
REVERSED
REWIND
REWRITE
RF
RH
RIGHT
ROUNDED
RUN

SA
SAME
SD
SEARCH
SECTION
SECURITY
SEEK
SEGMENT
SEGMENT-LIMIT
SELECT
SELECTED
SEND
SENTENCE
SEPARATE
SEQUENTIAL
SERVICE
SET
SIGN
SIZE
SKIP1
SKIP2
SKIP3
SORT
SORT-CORE-SIZE
SORT-FILE-SIZE
SORT-MERGE
SORT-MESSAGE
SORT-MODE-SIZE
SORT-RETURN
SOURCE
SOURCE-COMPUTER
SPACE
SPACES
SPECIAL-NAMES
STANDARD
START
STATUS
STOP
STRING
SUB-QUEUE-1
SUB-QUEUE-2
SUB-QUEUE-3
SUBTRACT
SUM
SUPERVISOR
SUPPRESS
SUSPEND
SYMBOLIC
SYNC
SYNCHRONIZED
SYSIN
SYSIPT
SYSLST
SYSOUT
SYSPCH
SYSPUNCH
SO1
SO2

TABLE
TALLY
TALLYING
TAPE
TERMINAL
TERMINATE
TEXT
THAN
THEN
THROUGH
THRU
TIME
TIME-OF-DAY
TIMES
TO
TOP
TOTALED
TOTALING
TRACE
TRACK
TRACK-AREA
TRACK-LIMIT
TRACKS
TRAILING
TRANSFORM
TYPE

UNEQUAL
UNIT
UNSTRING
UNTIL
UP
UPON
UPPER-BOUND
UPPER-BOUNDS
UPSI-0
UPSI-1
UPSI-2
UPSI-3
UPSI-4
UPSI-5
UPSI-6
UPSI-7
USAGE
USE
USING

VALUE
VALUES
VARYING

WHEN
WITH
WORDS
WORKING-STORAGE
WRITE
WRITE-ONLY
WRITE-VERIFY

ZERO
ZEROES
ZEROS

Appendix C
COBOL Language Summary: Abridged

General Format for IDENTIFICATION DIVISION

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  program-name.
[AUTHOR. [comment-entry] ... ]
[INSTALLATION. [comment-entry] ... ]
[DATE-WRITTEN. [comment-entry] ... ]
[DATE-COMPILED. [comment-entry] ... ]
[SECURITY. [comment-entry] ... ]
[REMARKS. [comment-entry] ... ]1
```

General Format for ENVIRONMENT DIVISION

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.  computer-name.
OBJECT-COMPUTER.  computer-name.
 [SPECIAL-NAMES.
     [implementor-name IS mnemonic] .... ].
 [INPUT-OUTPUT SECTION.
FILE-CONTROL.
     (file-control-entry)].
```

General Format for File Control Entry

```
SELECT file-name
  ASSIGN TO implementor-name
  [; ORGANIZATION IS SEQUENTIAL]
  [; ACCESS MODE IS SEQUENTIAL].
```

General Format for DATA DIVISION

```
 DATA DIVISION.
  [FILE SECTION.
   FD file-name
  [; BLOCK CONTAINS  integer-1  RECORDS]
    LABEL {RECORD IS   } {STANDARD}
          {RECORDS ARE } {OMITTED } .
  record-description-entry].
  [WORKING-STORAGE SECTION.
  record-description-entry ... ]
  [LINKAGE SECTION.
  record-description-entry ... ]
```

1. Although no longer part of COBOL, REMARKS is still recognized by many compilers.

General Format for Data Description Entry

Format 1

```
level-number {data-name-1}
             {FILLER     }
[; REDEFINES data-name-2]
[; {PICTURE} IS character-string]
   {PIC    }
[; USAGE IS {COMPUTATIONAL}]
            {COMP         }
            {DISPLAY      }
            {INDEX        }
[; OCCURS integer-2 TIMES
[ {ASCENDING } KEY IS data-name-4
  {DESCENDING}
   INDEXED BY index-name-1]
[; BLANK WHEN ZERO]
[; VALUE IS literal].
```

Format 2

```
88 condition-name {VALUE IS  } literal-1 [{THROUGH} literal-2] ...
                  {VALUES ARE}            [{THRU   }          ]
```

General Format for PROCEDURE DIVISION

```
PROCEDURE DIVISION  [USING data-name-1  [, data-name-2 ... ]].
[section-name SECTION].
[paragraph-name.  [sentence] ... ].
```

General Format for Verbs

```
ACCEPT identifier  [FROM mnemonic-name].
ACCEPT identifier FROM {DATE}
                       {DAY }
                       {TIME}
ADD {identifier-1} , {identifier-2}  ... TO identifier-m [ROUNDED]
    {literal-1   }   {literal-2   }
  [, identifier-n  [ROUNDED] ] ...
  [; ON SIZE ERROR imperative statement]
ADD {identifier-1}   , {identifier-2} , {identifier-3} ...
    {literal-1   }     {literal-2   }   {literal-3   }
  GIVING identifier-m  [ROUNDED] [, identifier-n [ROUNDED]] ...
  [; ON SIZE ERROR imperative statement].
ADD {CORRESPONDING} identifier-1 TO identifier-2 [ROUNDED]
    {CORR         }
  [; ON SIZE ERROR imperative-statement].
CALL identifier [USING data-name-1 [, data-name-2]  ... ].
CLOSE file-name-1 [, file-name-2   ... ].
COMPUTE identifier-1 [ROUNDED]
        [, identifier-2 [ROUNDED]    ... ]
  = arithmetic-expression [; ON SIZE ERROR imperative-statement.]
DISPLAY {identifier-1} , {identifier-2}  ...  [UPON mnemonic]
        {literal-1   }   {literal-2   }
```

```
DIVIDE {identifier-1} INTO identifier-2 [ROUNDED]
       {literal-1   }
  [, identifier-3 [ROUNDED]] ...
     [; ON SIZE ERROR imperative-statement].
DIVIDE {identifier-1} INTO {identifier-2}
       {literal-1   }      { literal-2  }
   GIVING identifier-3 [ROUNDED] [, identifier-4 [ROUNDED]]
      [; ON SIZE ERROR imperative-statement].
DIVIDE {identifier-1} BY {identifier-2}
       {literal-1   }    { literal-2  }
   GIVING identifier-3 [ROUNDED] [, identifier-4[ROUNDED]]
      [; ON SIZE ERROR imperative-statement].
GO TO  procedure-name-1.
IF condition {;THEN NEXT-SENTENCE}  {;ELSE NEXT-SENTENCE}
             { statement-1       }  { statement-2       }
MOVE {identifier-1} TO identifier-2 [, identifier-3] ...
     {literal-1   }
MOVE {CORRESPONDING} identifier-1 TO identifier-2.
     {CORR         }
MULTIPLY {identifier-1} BY identifier-2  [ROUNDED]
         {literal-1   }
   [, identifier-3 [ROUNDED]] ...
      [; ON SIZE ERROR imperative-statement].
MULTIPLY {identifier-1} BY {identifier-2}
         {literal-1   }    { literal-2  }
   GIVING identifier-3 [ROUNDED] [, identifier-4 [ROUNDED]]
      [; ON SIZE ERROR imperative-statement].
OPEN {INPUT } file-name-1 [, file-name-2] ...
     {OUTPUT}
PERFORM procedure-name.
PERFORM procedure-name {identifier} TIMES.
                       {integer   }
PERFORM procedure-name UNTIL condition-1
PERFORM procedure-name
   VARYING identifier-1 FROM  {identifier-2}
                              {literal-1   }
      BY {identifier-3} UNTIL  condition-1.
         {literal-2   }
READ file-name [INTO identifier]
   [; AT END imperative-statement].
SEARCH identifier-1  VARYING index-name-1
   [; AT END imperative-statement-1]
   ; WHEN condition-1 {imperative-statement-2} ...
                      {NEXT-SENTENCE         }

SEARCH ALL identifier-1
   [; AT END imperative-statement-1]
    ; WHEN data-name-1 {IS EQUAL TO}  {identifier-2               }
                       { IS =      }  {   literal-1               }
                                      {   arithmetic-expression-1 }
      [ AND data-name-2 {IS  EQUAL  TO}  {identifier-3           }     ]
      [                 {  IS =       }  {literal-2              } ... ]
      [                                  {arithmetic-expression-2}     ]
        {imperative-statement-2}
        {NEXT-SENTENCE         }
```

```
SET index-name-1 [, index-name-2] ...  TO {identifier-1 }
                                          {index-name-3 } .
                                          {integer-1    }
SET index-name-1 [, index-name-2] ... {UP BY  } {identifier-1.}
                                      {DOWN BY} {integer-1    }
STOP {RUN    }
     {literal}
SUBTRACT {identifier-1} , {identifier-2} ...
         {literal-1   }   {literal-2   }
    FROM identifier-m [ROUNDED] [, identifier-n [ROUNDED]] ...
       [; ON SIZE ERROR imperative-statement].
SUBTRACT {identifier-1} , {identifier-2} ...  FROM identifier-m
         {literal-1   }   {literal-2   }
     GIVING identifier-n [ROUNDED]
          [, identifier-o [ROUNDED]] ...
     [; ON SIZE ERROR imperative statement].
SUBTRACT {CORRESPONDING} identifier
         {CORR         }
     FROM identifier-2 [ROUNDED]
     [; ON SIZE ERROR imperative-statement].
WRITE record-name [FROM identifier-1]
   {BEFORE ADVANCING} {identifier-2 }
   {AFTER           } {integer      }
                      {mnemonic-name}
                      {PAGE         }
```

General Format for Conditions

```
RELATION CONDITION:
{arith-exp-1 } {IS [NOT] GREATER THAN} {identifier-2}
{literal-1   } {IS [NOT] LESS THAN   } {literal-2   }
{identifier-1} {IS [NOT] EQUAL TO    } {arith-exp-2 }
{index-name-1} {IS [NOT] >           } {index-name-2}
               {IS [NOT] <           }
               {IS [NOT] =           }
```

Class Condition

```
identifier IS [NOT] {NUMERIC   }
                    {ALPHABETIC}
```

Sign Condition

```
arithmetic-expression IS [NOT] {POSITIVE}
                               {NEGATIVE}
                               {ZERO    }
```

Condition-Name Condition

```
condition-name
```

Negated Simple Condition

```
NOT simple-condition
```

Combined Condition

```
condition {AND} condition        ...
          {OR }
```

Abbreviated Combined Relation Condition

```
relation-condition {AND} [NOT] [relational-operator] object        ...
                   {OR }
```

Miscellaneous Formats

Qualification

```
{condition-name} {OF} data-name-2
{data-name-1   } {IN}                ...
```

Subscripting

```
{condition-name} (subscript-1 [, subscript-2 [, subscript-3]])
{data-name     }
```

Identifier

```
data-name-1 {OF} data-name-2    ...
            {IN}
      [(subscript-1 [, subscript-2 [, subscript-3]])]
```

Glossary

A-Margin
Positions 8 through 11 of each line, or source record, in a COBOL program. Used to contain division, section, and paragraph entries.

Batch
The transactions that have been collected over a period of time for processing as a group.

Binary Search
A directed search that successively partitions a monotonically ordered table into smaller halves. By focusing the search upon the segment of the table that is likely to contain the item desired, if it is in the table at all, a significant part of the table is not examined.

Bi-stable Device
A COBOL variable that can take on one of two values. One value indicates that an anticipated condition has occurred. The other value indicates that the condition has not yet occurred.

B-Margin
Positions 12 through 72 of each line, or source record, in a COBOL program. Used to contain COBOL instructions that direct computer actions.

Buffer
In one context, the use of blanks to separate nonblank fields as a means of improving the reader's ability to comprehend the data contained in a computer-produced report. In another context, a staging, or holding, area in primary memory where record images are held before printing or immediately after reading.

Class Tests
Built-in COBOL functions that examine variables to see whether they are numeric (the numeric class test) or alphabetic (the alphabetic class test).

COBOL
(COmmon Business Oriented Language) A language in which a computer programmer may build computer programs for business applications.

Column Major Order
A process for establishing tables in primary memory so that all of the rows in column 1 come first, followed by all of the rows in column 2, followed by all of the rows in column 3, and so on.

Comments
The inclusion of English narrative as part of the COBOL source program to improve reader's comprehension of program functions.

Compound IF Statements
A device for exposing a field to multiple tests. It can be structured to avoid difficulties in interpretation and consequent program problems.

Computer-based Measurement Systems
The collection of computer-produced reports that indicate whether an organization's plans are being fulfilled.

Computer Clock
A hardware clock that maintains both the current time of day as well as the day of the year. These hardware-controlled values are accessible only through a request (i.e., ACCEPT) issued to the computer's operating system software.

Computer Programmer
The individual who constructs computer programs in a programming language, such as COBOL.

Condition Name
A symbolic name for a particular value that a variable can contain. When the variable contains the value specified, then the condition is said to exist.

Condition Tests
The use of a condition name (identified in an 88-level) in an IF statement to determine whether the condition exists.

Condition Variable
A symbolic name that represents more than one possible value a variable can contain. Condition names can be associated with only one value.

Control Break
A change in the value of a key, or important, field from one input record to the next.

Conventional-File Structures
The READ and WRITE verbs in COBOL. When these verbs are used, COBOL programs are much larger. Extra statements are the price paid for greater control over the input and output devices used by the program.

DATA DIVISION
The third of four divisions in every COBOL program. All field, variable, and record definitions are located within this part of the COBOL program.

Data Validity
The question in edit programs on whether or not a field contains the proper value.

Delimiter
A special character that marks both the start and the end of nonnumeric literals. In COBOL this character is a quote or an apostrophe.

EBCDIC
The Extended Binary Coded Decimal Interchange Code, which is the collating sequence of characters used to represent data in a computer.

Edit Programs
Programs that check the accuracy of data that has been transcribed from source documents to machine-readable form.

80–80 Listing
A report program that prints each 80-column record image read from the input file without reformatting the contents of the record.

ENVIRONMENT DIVISION
The second of four divisions to be found in every COBOL program.

Explicit Decimal Point
Inclusion of an actual decimal point that will appear in the printed version of the numeric field that contains this editing character.

External Module
Once again, the module contains a function better represented in isolated form. It is invoked by the use of a CALL verb and contains complete IDENTIFICATION, ENVIRONMENT, DATA, and PROCEDURE DIVISIONS.

Field
A collection of consecutive record positions that constitute a single item of data.

Field Access
A manner of defining an entire record so that any specific field within that record can have its contents retrieved or altered without disrupting the remainder of the record.

Field Validity Test
To examine a field that is usually part of a larger record structure to determine whether the data contained within is correct or incorrect.

FILLER
A special field name that informs the COBOL language processor that access to the field is not desired. The field contains data whose reference or use is not desired.

Fixed Dollar Sign
Placing a dollar sign to the left of the first digit in a numeric field whether or not that digit is a zero.

Good Programming Technique
The programmer who adds commentary as the program is being constructed and realizes that this effort is beneficial.

IDENTIFICATION DIVISION
The first of four divisions in every COBOL program. It serves to identify the circumstances that lead to the program's creation and use.

Implied Decimal Point
Informs the computer where a decimal point in a numeric field would be and, thus, identifies the integer and fractional portions of a numeric value.

Indentation
A programmer's device used to illustrate which program statements are subordinate to other statements.

Information Systems Analyst
The individual who designs computer-generated reports and the programs that produce those reports.

Initialized Variable
A field whose contents at the start of execution are known due to the use of the VALUE attribute in the field's definition.

Inquiry Statement
A statement that allows the programmer to determine the value a field contains. It is also known as a decision statement.

Internal File Name
The name immediately following SELECT. The name by which the file is referred throughout the PROCEDURE DIVISION.

Internal Module
A COBOL section in the Procedure Division invoked by use of a PERFORM verb. The section contains a function better represented as an isolated entry in the source program.

Invoke
To cause a subroutine to come into action and to carry out its assigned functions.

Iterative
An activity that occurs repeatedly for a fixed period of time.

Iterative Structure
The ability, when processing tables, to successively examine each entry in a multi-valued variable.

Key Field
An important field whose value, in the context of control breaks, indicates whether a new department or division has been encountered.

Leading Zero Suppression
The replacement of leading zeroes in a numeric value by specified characters.

Left Justification
When values are moved to nonnumeric fields they are started in the left-hand position of those fields.

Low-Volume I/O
The ACCEPT and DISPLAY verbs in COBOL. They are used when few record images are to be read and printed, respectively. The programmer sacrifices control in detecting end of file and printer control for a smaller program.

Mnemonic
A symbol that provides a COBOL name of the programmer's choosing for referring to a special function, like SYSIN or SYSOUT.

Model
A guide for building a computer program in which the actions of that program are already known. For example, edit program models differ from report program models due to their different function or purpose.

Multi-valued Variables
Variables that can hold, simultaneously, more than one value. Also known as vectors, tables, and/or matrices.

Nested IF Statements
A device placing IF statements inside other IF statements that is used to expose a field to multiple tests.

Nonnumeric Literal
A collection or sequence of values that is not and cannot be manipulated in an arithmetic operation.

Numeric Literals
A sequence of characters whose individual values are in the range of the numeric digits 0 through 9.

Numeric Variable
A COBOL field that contains only the digits 0 through 9 in each of its positions.

Operand
The numeric variables or literals acted upon by the arithmetic operator.

Operator
The arithmetic operation specified in the COBOL statement.

Operator Precedence
Determining which arithmetic operators are to act in what sequence, based upon the priorities of those arithmetic operators.

Page Overflow
A situation that occurs when printing the next record image will cause a line of printed output to appear in that portion of the page reserved for the trailers.

Paragraphs
A means by which sections can be partitioned into smaller, easier to understand units of COBOL programs.

Parentheses
A grouping device that specifies which part of arithmetic formulas are to be executed first, second, third, etc.

Point of Invocation
The location in the program where a subroutine is invoked. Usually, when the subroutine completes execution, control goes to the statement after the invocation.

Poor Programming Technique
The programmer who adds commentary to a program only after the program functions properly.

Printer Channels
The hardware device within the printer that allows the computing installation to control the processing of paper stock.

PROCEDURE DIVISION
The division composed of sections that consist of paragraphs that, in turn, contain executable instructions. The fourth of four divisions in every COBOL program.

Programming
A two-step process in which the programmer first determines the actions a computer must take to transform inputs to outputs, and second, translates those steps into a language such as COBOL.

Pseudo-Code
A shorthand, English-like language in which program logic can be expressed.

Qualification
The means through which variables whose names are not unique are identified by owner record structure.

Record Layouts
Layouts that describe the names, attributes, and physical location of the fields that constitute a record.

Record Structure
A collection of field definitions whose sum total equals a complete record for input or output data.

Relational Operator
The mechanism used in inquiry statements to specify how a variable and a value are to be related. For instance, is the variable equal to, less than, etc., a chosen literal value?

Report Programs
Programs that read, reformat, and then print selected records from an input file. Typically, the format of the printed report has been specified by the manager/client.

Reserved Words
COBOL verbs, such as ACCEPT, that cannot also be used in other contexts. Because of their special purposes, reserved words cannot be used as section, paragraph, or variable/field/record names.

Row Major Order
A process for establishing tables in primary memory so that all of the columns in row 1 come first, followed by all of the columns in row 2, followed by all of the columns in row 3, and so on.

Sections
A means by which related COBOL instructions can be identified according to program function.

Sequential Structure
An instruction that does not cause the flow of program control to change from its serial nature.

Sort Programs
Programs that sequence, or order, records in a file in a manner specified by the user of the program, usually provided by the vendor of the computer system.

Source Document
The document produced to record a transaction. It usually is not directly machine-readable and must first be converted to machine-readable form.

Subroutine
A collection of COBOL statements that, when required, are executed by a PERFORM instruction. These permit the construction of shorter programs by eliminating duplicate segments of coding.

Subscript
A numeric variable or literal that specifies the single entry within a multi-valued variable that we wish to examine or alter. The item is usually identified by its row/column number.

System Flowcharts
Graphic illustrations that name the input files used by a computer program as well as the output files produced by that program.

Temporary Fields
Variables in the DATA DIVISION required to hold the intermediate products of arithmetic formulas. Temporary fields are required because, when separate arithmetic verbs are used, a formula must be executed in parts, and the results of each part must be saved without destroying the contents of the original variables.

Titles
The printed lines that appear at the top of each page of the printed report.

Trailers
The printed lines that appear at the bottom of each page of the printed report.

Truncation
The loss of a character because a value larger than the field could hold was transferred to that field.

Uninitialized Variable
A field whose value at the start of execution is not known.

Unit Record Device
An I/O device that reads or writes unit records. Examples of these devices are card punches and line printers.

Update Programs
Programs that allow new records to be added, and existing records to be altered or deleted from files of related records (also known as file maintenance programs).

USAGE IS DISPLAY
A default attribute for all COBOL variables that specifies that the data in each variable are stored in human-readable form. Such data require conversion if it is to be involved in any arithmetic operation.

Variable
A symbolically named collection of primary memory locations into which data may be stored and subsequently retrieved. (It should be noted that records consist of variables.)

Index

wcb
Wm. C. Brown Company Publishers
Dubuque, Iowa
ISBN 0-697-08149-4